PennOS

1

Generated by Doxygen 1.9.1

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 directory_entry Struct Reference

`#include <fat.h>`

Data Fields

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

3.1.1 Field Documentation

3.1.1.1 _BUFFER_

`char directory_entry::_BUFFER_[16]`

3.1.1.2 firstBlock

`uint16_t directory_entry::firstBlock`

3.1.1.3 mtime

```
time_t directory_entry::mtime
```

3.1.1.4 name

```
char directory_entry::name[32]
```

3.1.1.5 perm

```
uint8_t directory_entry::perm
```

3.1.1.6 size

```
uint32_t directory_entry::size
```

3.1.1.7 type

```
uint8_t directory_entry::type
```

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

- src/pennfat/fat.h

3.2 file Struct Reference

```
#include <filesystem.h>
```

Collaboration diagram for file:

Data Fields

- char filename [32]
- int file_id
- int wr_pid
- struct fileptr * fileptr_head
- struct file * next

3.2.1 Field Documentation

3.2.1.1 file_id

```
int file::file_id
```

3.2.1.2 filename

```
char file::filename[32]
```

3.2.1.3 fileptr_head

```
struct fileptr* file::fileptr_head
```

3.2.1.4 next

```
struct file* file::next
```

3.2.1.5 wr_pid

```
int file::wr_pid
```

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

- src/filesystem/filesystem.h

3.3 fileptr Struct Reference

```
#include <filesystem.h>
```

Collaboration diagram for fileptr:

Data Fields

- int pid
- int ptr
- fileptr_t * next

3.3.1 Field Documentation

3.3.1.1 next

```
fileptr_t* fileptr::next
```

3.3.1.2 pid

```
int fileptr::pid
```

3.3.1.3 ptr

```
int fileptr::ptr
```

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

- src/filesystem/filesystem.h

3.4 job Struct Reference

```
#include <job-list.h>
```

Collaboration diagram for job:

Data Fields

- int job_id
- int pid
- int stop_order
- bool done
- job_t * next

3.4.1 Field Documentation

3.4.1.1 done

```
bool job::done
```

3.4.1.2 job_id

```
int job::job_id
```

3.4.1.3 next

```
job_t* job::next
```

3.4.1.4 pid

```
int job::pid
```

3.4.1.5 stop_order

```
int job::stop_order
```

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

- src/shell/job-list.h

3.5 parsed_command Struct Reference

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

Data Fields

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

3.5.1 Detailed Description

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

3.5.2 Field Documentation

3.5.2.1 commands

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

3.5.2.2 is_background

```
bool parsed_command::is_background
```

3.5.2.3 is_file_append

```
bool parsed_command::is_file_append
```

3.5.2.4 num_commands

```
size_t parsed_command::num_commands
```

3.5.2.5 stdin_file

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

3.5.2.6 stdout_file

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

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

- src/util/parser.h

3.6 PCB Struct Reference

```
#include <PCB.h>
```

Collaboration diagram for PCB:

Data Fields

- char * name
- ucontext_t * context
- pid_t parent_pid
- pid_t pid
- pid_t children [10000]
- int numChildren
- int fileDescriptors [MAX_FDS]
- int priority
- int status
- struct PCB * next

3.6.1 Field Documentation

3.6.1.1 children

```
pid_t PCB::children[10000]
```

3.6.1.2 context

```
ucontext_t* PCB::context
```

3.6.1.3 fileDescriptors

```
int PCB::fileDescriptors[MAX_FDS]
```

3.6.1.4 name

```
char* PCB::name
```

3.6.1.5 next

```
struct PCB* PCB::next
```

3.6.1.6 numChildren

```
int PCB::numChildren
```

3.6.1.7 parent_pid

```
pid_t PCB::parent_pid
```

3.6.1.8 pid

```
pid_t PCB::pid
```

3.6.1.9 priority

```
int PCB::priority
```

3.6.1.10 status

```
int PCB::status
```

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

- src/kernel/PCB.h

3.7 point Struct Reference

```
#include <fat.h>
```

Data Fields

- int first
- int second

3.7.1 Field Documentation

3.7.1.1 first

```
int point::first
```

3.7.1.2 second

```
int point::second
```

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

- src/pennfat/fat.h

Chapter 4

File Documentation

4.1 src/filesystem/filesystem.c File Reference

This is a brief description of the contents of the file.

```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include "filesystem.h"
#include "../util/globals.h"
#include "../util/util.h"
#include "../util/p-errno.h"
#include "../kernel/PCB.h"
#include "../pennfat/fat.h"
#include "../pennfat/safe.h"
```

Include dependency graph for filesystem.c:

Macros

- #define MIN(a, b) (((a) < (b)) ? (a) : (b))
- #define BETWEEN_INCL(value, lower, upper) ((value) >= (lower) && (value) <= (upper))
- #define F_HASPERM(perm, mask) (((perm) & (mask)) != 0)
- #define F_CANREAD(perm) (F_HASPERM(perm, FILEPERM_RD))
- #define F_CANWRITE(perm) ((F_HASPERM(perm, FILEPERM_WR)) && (F_CANREAD(perm)))

Functions

- void create_fileptr (fileptr_t **fileptr_head, int pid, int ptr)

  create & insert a file pointer into the fileptr list
- void delete_fileptr (fileptr_t **fileptr_head, int pid)
- fileptr_t * get_fileptr (fileptr_t *fileptr_head, int pid)
- int get_fileptr_ptr (fileptr_t *fileptr_head, int pid)
- int create_file_entry (const char *filename, int mode, dir_entry_t *dir_entry)
- void delete_file_entry (int file_id)
- bool valid_perm (uint8_t perm, int mode)
- file_t * find_file_entry_by_file_id (int file_id)
- file_t * find_file_entry_by_filename (const char *filename)

- bool find_file_entry (int fd, int *file_id_ptr, file_t *file_entry)
- void process_create_fileptrs (PCB *pcb)
- void process_delete_fileptrs (PCB *pcb)
- void print_fileptr_pids_all ()
- int find_unused_fd (PCB *pcb)
- bool is_duplicate_fd (PCB *pcb, int file_id)
- bool f_isatty (int fd)
- int f_open (const char *fname, int mode)

  Opens or creates a file and returns a file descriptor.
- int f_read (int fd, int n, char *buf)

  Reads data from a file descriptor.
- int f_write (int fd, const char *str, int n)

  Writes data to a file descriptor.
- int f_close (int fd)

  Closes a file descriptor.
- int f_unlink (const char *fname)

  Unlinks (deletes) a file.
- int f_lseek (int fd, int offset, int whence)

  Moves the file pointer to a specified position within a file.
- void f_ls (const char *filename)

  Lists information about files in the file system.
- void f_touch (char *filenames[ ], int n)
- int f_print (const char *str)

  Prints a string to the standard error (stderr).
- int f_mount (char *fs_name, uint16_t **fat)

  Mounts a file system.
- void f_unmount (uint16_t **fat, int fs_fd)

  Unmounts a file system.
- void f_mv (char *src, char *dest)

  Moves or renames a file or directory.
- void f_cp (char *src, char *dest)

  Copies a file or directory.
- void f_rm (char *filenames[ ], int n)

  Removes (deletes) files or directories.
- void f_chmod (char *filename, int perms)

  Changes the permissions of a file or directory.

Variables

- int next_file_id = 0
- file_t * open_files = NULL
- PCB * current_pcb

4.1.1 Detailed Description

This is a brief description of the contents of the file.

4.1.2 Macro Definition Documentation

4.1.2.1 BETWEEN_INCL

```
#define BETWEEN_INCL(
              value,
              lower,
              upper ) ((value) >= (lower) && (value) <= (upper))
```

4.1.2.2 F_CANREAD

```
#define F_CANREAD(
              perm ) (F_HASPERM(perm, FILEPERM_RD))
```

4.1.2.3 F_CANWRITE

```
#define F_CANWRITE(
              perm ) ((F_HASPERM(perm, FILEPERM_WR)) && (F_CANREAD(perm)))
```

4.1.2.4 F_HASPERM

```
#define F_HASPERM(
              perm,
              mask ) (((perm) & (mask)) != 0)
```

4.1.2.5 MIN

```
#define MIN(
              a,
              b ) (((a) < (b)) ?  (a) :  (b))
```

4.1.3 Function Documentation

4.1.3.1 create_file_entry()

```
int create_file_entry (
             const char * filename,
             int mode,
             dir_entry_t * dir_entry )
```

create & insert a file entry into `open_files`

Parameters

filename	the file name
mode	the open mode
dir_entry	the file's fat directory entry

Returns

the file_id

4.1.3.2 create_fileptr()

```
void create_fileptr (
            fileptr_t ** fileptr_head,
            int pid,
            int ptr )
```

create & insert a file pointer into the fileptr list

Parameters

fileptr_head	the head of the fileptr list
pid	the process id for the fileptr
ptr	the pointer/offset

Returns

none

4.1.3.3 delete_file_entry()

```
void delete_file_entry (
            int file_id )
```

delete the specified file entry from `open_files`; call when `file_entry.fileptr_head == NULL`

Parameters

file_id	the file id

Returns

none

4.1.3.4 delete_fileptr()

```
void delete_fileptr (
            fileptr_t ** fileptr_head,
            int pid )
```

delete the specified file pointer from the fileptr list

Parameters

fileptr_head	the head of the fileptr list
pid	the process id whose fileptr should be deleted

Returns

none

4.1.3.5 f_chmod()

```
void f_chmod (
            char * filename,
            int perms )
```

Changes the permissions of a file or directory.

Parameters

filename	The name of the file or directory.
perms	The new permissions to set for the file or directory.

4.1.3.6 f_close()

```
int f_close (
            int fd )
```

Closes a file descriptor.

This function closes the specified file descriptor, releasing associated resources. If the file descriptor represents a terminal, an error is returned. The function then finds the corresponding file entry in open files, updates the process's file descriptor table, and deallocates file pointers if this is the last instance of the file descriptor for the process. If the process had write access, it is revoked when the last instance is closed. If no process is using the file, the file entry is deleted.

Parameters

fd	The file descriptor to close.

Returns

On success, returns 0. On failure, returns -1, and the global variable ERRNO is set accordingly.

4.1.3.7 f_cp()

```
void f_cp (
            char * src,
            char * dest )
```

Copies a file or directory.

Parameters

src	The source path of the file or directory.
dest	The destination path for the copied file or directory.

4.1.3.8 f_isatty()

```
bool f_isatty (
            int fd )
```

check whether a fd refers to the terminal

Parameters

fd	the file descriptor

Returns

`true` if `fd` is either `F_STDIN`, `F_STDOUT`, or `F_STDERR`; `false` otherwise

4.1.3.9 f_ls()

```
void f_ls (
            const char * filename )
```

Lists information about files in the file system.

If the given filename is NULL, this function lists information about all files in the file system. Otherwise, it lists information about the specified file. It locates the file entry, and if found, displays information about the file using the fs_ls_single function. If the file is not found, ERR_FS_FILE_NOT_FOUND is set in ERRNO.

Parameters

filename	The name of the file to list information about. If NULL, lists information about all files.

4.1.3.10 f_lseek()

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

Moves the file pointer to a specified position within a file.

This function adjusts the file pointer for the specified file descriptor, allowing seeking to a new position within the file. If the file descriptor represents a terminal, an error is returned. The function then locates the corresponding file entry, retrieves the current file pointer position, and calculates the new offset based on the specified 'whence' parameter. If the new offset is within the file bounds, the file pointer is updated, and the new position is returned. Otherwise, an error is returned.

Parameters

fd	The file descriptor to seek within.
offset	The offset to move the file pointer.
whence	The reference position for the offset (F_SEEK_CURR, F_SEEK_END, or F_SEEK_SET).

Returns

On success, returns the new file pointer position. On failure, returns -1, and the global variable ERRNO is set accordingly.

4.1.3.11 f_mount()

```
int f_mount (
             char * fs_name,
             uint16_t ** fat )
```

Mounts a file system.

Parameters

fs_name	The name of the file system to mount.
fat	A pointer to the file allocation table (FAT) array.

Returns

On success, returns 0. On failure, returns -1.

4.1.3.12 f_mv()

```
void f_mv (
            char * src,
            char * dest )
```

Moves or renames a file or directory.

Parameters

src	The source path of the file or directory.
dest	The destination path for the file or directory.

4.1.3.13 f_open()

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

Opens or creates a file and returns a file descriptor.

Parameters

fname	The name of the file to open or create.
mode	The mode in which to open the file (F_READ, F_WRITE, or F_APPEND).

Returns

The file descriptor on success, or -1 on failure with ERRNO set.

Note

If the file already exists, the function checks permissions and handles multiple processes attempting to open the same file.

If the file does not exist, it is created, and the open files list is updated.

4.1.3.14 f_print()

```
int f_print (
            const char * str )
```

Prints a string to the standard error (stderr).

Parameters

str	The string to be printed.

Returns

On success, returns the number of bytes written. On failure, returns -1.

4.1.3.15 f_read()

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

Reads data from a file descriptor.

This function reads data from the specified file descriptor and stores it in the provided buffer. If the file descriptor represents STDIN, data is read from the terminal input. If it represents STDOUT or STDERR, an error is returned. For regular file descriptors, the corresponding file entry is located, and data is read from the file into a temporary buffer.

Parameters

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

Returns

On success, the number of bytes read is returned. On failure, -1 is returned, and the global variable ERRNO is set accordingly. If the end of the file is reached (EOF), 0 is returned.

4.1.3.16 f_rm()

```
void f_rm (
            char * filenames[],
            int n )
```

Removes (deletes) files or directories.

Parameters

filenames	An array of strings containing the names of the files or directories to be removed.
n	The number of filenames in the array.

4.1.3.17 f_touch()

```
void f_touch (
            char * filenames[],
            int n )
```

Creates empty files with the specified names.

Parameters

filenames	An array of strings containing the names of the files to be created.
n	The number of filenames in the array.

4.1.3.18 f_unlink()

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

Unlinks (deletes) a file.

This function unlinks (deletes) the specified file, releasing associated resources. It first locates the file entry by filename and checks if the current process is accessing the file. If so, it removes the file pointer and updates the write access status. If no more processes are accessing the file, the file entry is deleted, and the file system is updated. If the file is still in use by another process, it is marked as deleted in the file system.

Parameters

fname	The name of the file to unlink.

Returns

On success, returns 0. On failure, returns -1, and the global variable ERRNO is set accordingly.

4.1.3.19 f_unmount()

```
void f_unmount (
            uint16_t ** fat,
            int fs_fd )
```

Unmounts a file system.

Parameters

fat	A pointer to the file allocation table (FAT) array.
fs↩ _fd	The file descriptor of the file system to unmount.

4.1.3.20 f_write()

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

Writes data to a file descriptor.

This function writes data to the specified file descriptor based on the given parameters. If the file descriptor represents STDOUT or STDERR, the data is output to the terminal. For regular file descriptors, the function locates the corresponding file entry, checks for write access, and updates the file content accordingly.

Parameters

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

Returns

On success, returns the actual number of bytes written. On failure, returns -1, and the global variable ERRNO is set accordingly.

4.1.3.21 find_file_entry()

```
bool find_file_entry (
            int fd,
            int * file_id_ptr,
            file_t * file_entry )
```

find a file by `fd`

Parameters

file_id_ptr	set to the file id, if the file is found
file_entry	set to the file entry, if the file is found

Returns

`true` if the file is found, `false` & print otherwise

4.1.3.22 find_file_entry_by_file_id()

```
file_t* find_file_entry_by_file_id (
            int file_id )
```

4.1.3.23 find_file_entry_by_filename()

```
file_t* find_file_entry_by_filename (
            const char * filename )
```

4.1.3.24 find_unused_fd()

```
int find_unused_fd (
            PCB * pcb )
```

get the first unused file descriptor (for `f_open`)

Parameters

pcb	the calling process

Returns

the first unused fd, or `-1` if the fd table is full

4.1.3.25 get_fileptr()

```
fileptr_t* get_fileptr (
            fileptr_t * fileptr_head,
            int pid )
```

get a file pointer struct for a process

Parameters

fileptr_head	the head of the fileptr list
pid	the process id to find a fileptr for

Returns

the file pointer struct, or `NULL` if `pid` was not found

4.1.3.26 get_fileptr_ptr()

```
int get_fileptr_ptr (
            fileptr_t * fileptr_head,
            int pid )
```

get a file pointer for a process

Parameters

fileptr_head	the head of the fileptr list
pid	the process id to find a fileptr for

Returns

the file pointer, or `-1` if `pid` was not found

4.1.3.27 is_duplicate_fd()

```
bool is_duplicate_fd (
            PCB * pcb,
            int file_id )
```

check if a process already has the file open; use this to check whether to create a file pointer

Parameters

pcb	the process PCB
file↩ _id	the file id (from `file_t.file_id`)

Returns

`true` if `file_id` is already in the file descriptor table of `pcb`, `false` otherwise

4.1.3.28 print_fileptr_pids_all()

```
void print_fileptr_pids_all ( )
```

DEBUG: print all file pointers & their pids

Returns

none

4.1.3.29 process_create_fileptrs()

```
void process_create_fileptrs (
            PCB * pcb )
```

create file pointers for each unique file_id (called by `p_spawn`)

Parameters

pcb	the process PCB

Returns

none

4.1.3.30 process_delete_fileptrs()

```
void process_delete_fileptrs (
            PCB * pcb )
```

delete file pointers of each unique file_id (called by `p_kill`)

Parameters

pcb	the process PCB

Returns

none

4.1.3.31 valid_perm()

```
bool valid_perm (
            uint8_t perm,
            int mode )
```

check if permissions requested match a file's allowed permissions

Parameters

perm	the allowed permissions
mode	the permissions requested

Returns

`true` if `perm` allows a file to be opened with `mode`, `false` otherwise

4.1.4 Variable Documentation

4.1.4.1 current_pcb

```
PCB* current_pcb  [extern]
```

4.1.4.2 next_file_id

```
int next_file_id = 0
```

4.1.4.3 open_files

```
file_t* open_files = NULL
```

4.2 src/filesystem/filesystem.h File Reference

```
#include <stdint.h>
#include "../kernel/PCB.h"
```

Include dependency graph for filesystem.h: This graph shows which files directly or indirectly include this file:

Data Structures

- struct fileptr
- struct file

Macros

- #define F_STDIN 0
- #define F_STDOUT 1
- #define F_STDERR 2
- #define F_WRITE 0
- #define F_READ 1
- #define F_APPEND 2
- #define F_SEEK_SET 0
- #define F_SEEK_CURR 1
- #define F_SEEK_END 2
- #define FPERM_READ 0b100
- #define FPERM_WRIT 0b010
- #define FPERM_EXEC 0b001

Typedefs

- typedef struct fileptr fileptr_t
- typedef struct file file_t

Functions

- int f_open (const char *fname, int mode)

  Opens or creates a file and returns a file descriptor.
- int f_read (int fd, int n, char *buf)

  Reads data from a file descriptor.
- int f_write (int fd, const char *str, int n)

  Writes data to a file descriptor.
- int f_close (int fd)

  Closes a file descriptor.
- int f_unlink (const char *fname)

  Unlinks (deletes) a file.
- int f_lseek (int fd, int offset, int whence)

  Moves the file pointer to a specified position within a file.
- void f_ls (const char *filename)

  Lists information about files in the file system.
- void f_touch (char *filenames[ ], int n)
- int f_print (const char *str)

  Prints a string to the standard error (stderr).
- int f_mount (char *fs_name, uint16_t **fat)

  Mounts a file system.
- void f_unmount (uint16_t **fat, int fs_fd)

  Unmounts a file system.
- void f_mv (char *src, char *dest)

  Moves or renames a file or directory.
- void f_cp (char *src, char *dest)

  Copies a file or directory.
- void f_rm (char *filenames[ ], int n)

  Removes (deletes) files or directories.
- void f_chmod (char *filename, int perms)

  Changes the permissions of a file or directory.
- void print_fileptr_pids_all ()
- void process_create_fileptrs (PCB *pcb)
- void process_delete_fileptrs (PCB *pcb)
- file_t * find_file_entry_by_file_id (int file_id)

4.2.1 Macro Definition Documentation

4.2.1.1 F_APPEND

```
#define F_APPEND 2
```

4.2.1.2 F_READ

```
#define F_READ 1
```

4.2.1.3 F_SEEK_CURR

```
#define F_SEEK_CURR 1
```

4.2.1.4 F_SEEK_END

```
#define F_SEEK_END 2
```

4.2.1.5 F_SEEK_SET

```
#define F_SEEK_SET 0
```

4.2.1.6 F_STDERR

```
#define F_STDERR 2
```

4.2.1.7 F_STDIN

```
#define F_STDIN 0
```

4.2.1.8 F_STDOUT

```
#define F_STDOUT 1
```

4.2.1.9 F_WRITE

```
#define F_WRITE 0
```

4.2.1.10 FPERM_EXEC

```
#define FPERM_EXEC 0b001
```

4.2.1.11 FPERM_READ

```
#define FPERM_READ 0b100
```

4.2.1.12 FPERM_WRIT

```
#define FPERM_WRIT 0b010
```

4.2.2 Typedef Documentation

4.2.2.1 file_t

```
typedef struct file file_t
```

4.2.2.2 fileptr_t

```
typedef struct fileptr fileptr_t
```

4.2.3 Function Documentation

4.2.3.1 f_chmod()

```
void f_chmod (
            char * filename,
            int perms )
```

Changes the permissions of a file or directory.

Parameters

filename	The name of the file or directory.
perms	The new permissions to set for the file or directory.

4.2.3.2 f_close()

```
int f_close (
            int fd )
```

Closes a file descriptor.

close a file descriptor

Parameters

fd	the file descriptor to close

Returns

`0` on success, `-1` on error

This function closes the specified file descriptor, releasing associated resources. If the file descriptor represents a terminal, an error is returned. The function then finds the corresponding file entry in open files, updates the process's file descriptor table, and deallocates file pointers if this is the last instance of the file descriptor for the process. If the process had write access, it is revoked when the last instance is closed. If no process is using the file, the file entry is deleted.

Parameters

fd	The file descriptor to close.

Returns

On success, returns 0. On failure, returns -1, and the global variable ERRNO is set accordingly.

4.2.3.3 f_cp()

```
void f_cp (
            char * src,
            char * dest )
```

Copies a file or directory.

Parameters

src	The source path of the file or directory.
dest	The destination path for the copied file or directory.

4.2.3.4 f_ls()

```
void f_ls (
            const char * filename )
```

Lists information about files in the file system.

list a file in the current directory

Parameters

filename	the file to list, or `NULL` to list all files in the current directory

Returns

none

If the given filename is NULL, this function lists information about all files in the file system. Otherwise, it lists information about the specified file. It locates the file entry, and if found, displays information about the file using the fs_ls_single function. If the file is not found, ERR_FS_FILE_NOT_FOUND is set in ERRNO.

Parameters

filename	The name of the file to list information about. If NULL, lists information about all files.

4.2.3.5 f_lseek()

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

Moves the file pointer to a specified position within a file.

reposition the file pointer

Parameters

fd	the file
offset	the offset
whence	Either `F_SEEK_SET`, `F_SEEK_CURR`, or `F_SEEK_END`. If `F_SEEK_SET`: file pointer is set to `offset`. If `F_SEEK_CURR`: file pointer is set to `offset` + the current file pointer position. If `F_SEEK_END`: file pointer is set to `offset` + file size.

Returns

the new file pointer position, or -1 on error

This function adjusts the file pointer for the specified file descriptor, allowing seeking to a new position within the file. If the file descriptor represents a terminal, an error is returned. The function then locates the corresponding file entry, retrieves the current file pointer position, and calculates the new offset based on the specified 'whence' parameter. If the new offset is within the file bounds, the file pointer is updated, and the new position is returned. Otherwise, an error is returned.

Parameters

fd	The file descriptor to seek within.
offset	The offset to move the file pointer.
whence	The reference position for the offset (F_SEEK_CURR, F_SEEK_END, or F_SEEK_SET).

Returns

On success, returns the new file pointer position. On failure, returns -1, and the global variable ERRNO is set accordingly.

4.2.3.6 f_mount()

```
int f_mount (
              char * fs_name,
              uint16_t ** fat )
```

Mounts a file system.

Parameters

fs_name	The name of the file system to mount.
fat	A pointer to the file allocation table (FAT) array.

Returns

On success, returns 0. On failure, returns -1.

4.2.3.7 f_mv()

```
void f_mv (
              char * src,
              char * dest )
```

Moves or renames a file or directory.

Parameters

src	The source path of the file or directory.
dest	The destination path for the file or directory.

4.2.3.8 f_open()

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

Opens or creates a file and returns a file descriptor.

open a file name fname with the mode `mode` and return a file descriptor

Parameters

fname	the filename to open
mode	Either `F_WRITE`, `F_READ`, or `F_APPEND`. If `F_WRITE`: reading and writing, truncate if the file exists and create it otherwise; only 1 file instance of `F_WRITE` mode can exist. If `F_READ`: read only. If `F_APPEND`: reading and writing, do not truncate if the file exists; file pointer is set to end of file.

Returns

a file descriptor on success, `-1` otherwise

Parameters

fname	The name of the file to open or create.
mode	The mode in which to open the file (F_READ, F_WRITE, or F_APPEND).

Returns

The file descriptor on success, or -1 on failure with ERRNO set.

Note

If the file already exists, the function checks permissions and handles multiple processes attempting to open the same file.

If the file does not exist, it is created, and the open files list is updated.

4.2.3.9 f_print()

```
int f_print (
            const char * str )
```

Prints a string to the standard error (stderr).

print to terminal (F_STDERR)

Parameters

str	the string to print (use `snprintf` to format)

Returns

number of bytes written (including `\0`) on success, `-1` on error

Parameters

str	The string to be printed.

Returns

On success, returns the number of bytes written. On failure, returns -1.

4.2.3.10 f_read()

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

Reads data from a file descriptor.

read from a file

Parameters

fd	the file descriptor to read from
n	number of bytes to read
buf	buffer to read into

Returns

number of bytes read (including `\0`) on success, `0` if EOF is reached, `-1` on error

This function reads data from the specified file descriptor and stores it in the provided buffer. If the file descriptor represents STDIN, data is read from the terminal input. If it represents STDOUT or STDERR, an error is returned. For regular file descriptors, the corresponding file entry is located, and data is read from the file into a temporary buffer.

Parameters

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

Returns

On success, the number of bytes read is returned. On failure, -1 is returned, and the global variable ERRNO is set accordingly. If the end of the file is reached (EOF), 0 is returned.

4.2.3.11 f_rm()

```
void f_rm (
            char * filenames[],
            int n )
```

Removes (deletes) files or directories.

Parameters

filenames	An array of strings containing the names of the files or directories to be removed.
n	The number of filenames in the array.

4.2.3.12 f_touch()

```
void f_touch (
            char * filenames[],
            int n )
```

list a file in the current directory

Parameters

filenames	files to touch
n	number of files to touch

Returns

none

Creates empty files with the specified names.

Parameters

filenames	An array of strings containing the names of the files to be created.
n	The number of filenames in the array.

4.2.3.13 f_unlink()

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

Unlinks (deletes) a file.

remove a file, if it exists & is not in use

Parameters

fname	the file to remove

Returns

`0` on success, `-1` otherwise

This function unlinks (deletes) the specified file, releasing associated resources. It first locates the file entry by filename and checks if the current process is accessing the file. If so, it removes the file pointer and updates the write access status. If no more processes are accessing the file, the file entry is deleted, and the file system is updated. If the file is still in use by another process, it is marked as deleted in the file system.

Parameters

fname	The name of the file to unlink.

Returns

On success, returns 0. On failure, returns -1, and the global variable ERRNO is set accordingly.

4.2.3.14 f_unmount()

```
void f_unmount (
            uint16_t ** fat,
            int fs_fd )
```

Unmounts a file system.

Parameters

fat	A pointer to the file allocation table (FAT) array.
fs_fd	The file descriptor of the file system to unmount.

4.2.3.15 f_write()

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

Writes data to a file descriptor.

write to a file & increment file pointer

Parameters

fd	the file descriptor to write to
str	the string to write from
n	number of bytes to write

Returns

number of bytes written (including \0) on success, -1 on error

This function writes data to the specified file descriptor based on the given parameters. If the file descriptor represents STDOUT or STDERR, the data is output to the terminal. For regular file descriptors, the function locates the corresponding file entry, checks for write access, and updates the file content accordingly.

Parameters

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

Returns

On success, returns the actual number of bytes written. On failure, returns -1, and the global variable ERRNO is set accordingly.

4.2.3.16 find_file_entry_by_file_id()

```
file_t* find_file_entry_by_file_id (
            int file_id )
```

4.2.3.17 print_fileptr_pids_all()

```
void print_fileptr_pids_all ( )
```

DEBUG: print all file pointers & their pids

Returns

none

4.2.3.18 process_create_fileptrs()

```
void process_create_fileptrs (
            PCB * pcb )
```

create file pointers for each unique file_id (called by `p_spawn`)

Parameters

pcb	the process PCB

Returns

none

4.2.3.19 process_delete_fileptrs()

```
void process_delete_fileptrs (
            PCB * pcb )
```

delete file pointers of each unique file_id (called by `p_kill`)

Parameters

pcb	the process PCB

Returns

none

4.3 src/kernel/kernel-functions.c File Reference

```
#include "PCB.h"
#include "kernel-functions.h"
#include "../logger/logger.h"
#include "../util/globals.h"
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include "stdio.h"
```

Include dependency graph for kernel-functions.c:

Functions

- PCB * k_process_create (PCB *parent)
- int k_process_kill (PCB *process, int signal)
- void k_process_deep_cleanup (PCB *process)
- void k_process_cleanup (PCB *process)

4.3.1 Function Documentation

4.3.1.1 k_process_cleanup()

```
void k_process_cleanup (
            PCB * process )
```

frees PCB `process`

Parameters

process	pointer of PCB to be freed

Returns

none

4.3.1.2 k_process_create()

```
PCB* k_process_create (
            PCB * parent )
```

creates PCB and adds it to global PCB list

Parameters

parent	the parent of PCB to be created, if it exists

Returns

created PCB

4.3.1.3 k_process_deep_cleanup()

```
void k_process_deep_cleanup (
            PCB * process )
```

frees PCB `process` and all of its descendants

Parameters

process	pointer of PCB (and its descendants) to be freed

Returns

none

4.3.1.4 k_process_kill()

```
int k_process_kill (
            PCB * process,
            int signal )
```

sends signal `signal` to inputted PCB `process`

Parameters

process	pointer of PCB to send signal to
signal	signal to send

Returns

0 on success; 1 on failure

4.4 src/kernel/kernel-functions.h File Reference

`#include "PCB.h"`
Include dependency graph for kernel-functions.h: This graph shows which files directly or indirectly include this file:

Functions

- PCB * k_process_create (PCB *parent)
- int k_process_kill (PCB *process, int signal)
- void k_process_deep_cleanup (PCB *process)
- void k_process_cleanup (PCB *process)

4.4.1 Function Documentation

4.4.1.1 k_process_cleanup()

```
void k_process_cleanup (
            PCB * process )
```

frees PCB `process`

Parameters

process	pointer of PCB to be freed

Returns

none

4.4.1.2 k_process_create()

```
PCB* k_process_create (
            PCB * parent )
```

creates PCB and adds it to global PCB list

Parameters

parent	the parent of PCB to be created, if it exists

Returns

created PCB

4.4.1.3 k_process_deep_cleanup()

```
void k_process_deep_cleanup (
            PCB * process )
```

frees PCB `process` and all of its descendants

Parameters

process	pointer of PCB (and its descendants) to be freed

Returns

none

4.4.1.4 k_process_kill()

```
int k_process_kill (
            PCB * process,
            int signal )
```

sends signal `signal` to inputted PCB `process`

Parameters

process	pointer of PCB to send signal to
signal	signal to send

Returns

0 on success; 1 on failure

4.5 src/kernel/PCB.c File Reference

```
#include "PCB.h"
#include "scheduler.h"
#include <stdio.h>
#include "../util/globals.h"
#include <valgrind/valgrind.h>
```

Include dependency graph for PCB.c:

4.6 src/kernel/PCB.h File Reference

```
#include <ucontext.h>
#include <sys/types.h>
#include <stdbool.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
```

Include dependency graph for PCB.h: This graph shows which files directly or indirectly include this file:

Data Structures

- struct PCB

Macros

- #define STACKSIZE 4096 * 256
- #define MAX_FDS 1024
- #define NOFILE -1
- #define STDIN_ID -2
- #define STDOUT_ID -3
- #define STDERR_ID -4

Typedefs

- typedef struct PCB PCB

Functions

- void k_free (PCB *pcb)
- PCB * createPCB (PCB *parent)
- void addPCBToList (PCB **list, PCB *pcb)
- void removePCBFromList (PCB **list, PCB *pcb)
- PCB * findPCBByPID (pid_t pid)
- PCB * findPCBByContext (ucontext_t *context)
- int getLength (PCB *list)
- int count_running (PCB *head)
- int count_running_priority (PCB *head, int prio)

Variables

- PCB * pcb_list
- pid_t next_pid

4.6.1 Macro Definition Documentation

4.6.1.1 MAX_FDS

```
#define MAX_FDS 1024
```

4.6.1.2 NOFILE

```
#define NOFILE -1
```

4.6.1.3 STACKSIZE

```
#define STACKSIZE 4096 * 256
```

4.6.1.4 STDERR_ID

```
#define STDERR_ID -4
```

4.6.1.5 STDIN_ID

```
#define STDIN_ID -2
```

4.6.1.6 STDOUT_ID

```
#define STDOUT_ID -3
```

4.6.2 Typedef Documentation

4.6.2.1 PCB

```
typedef struct PCB PCB
```

4.6.3 Function Documentation

4.6.3.1 addPCBToList()

```
void addPCBToList (
            PCB ** head,
            PCB * pcb )
```

add a process to the PCBList

Parameters

list	the PCBList
pcb	the process PCB to add

Returns

none

adds a given a PCB pcb to list

Parameters

head	pointer to head of circular linked list
pcb	the pcb we want to add

Returns

None

4.6.3.2 count_running()

```
int count_running (
            PCB * head )
```

count number of T_RUNNING processes in a circular linked list

Parameters

head	pointer to the head of circular linked list

Returns

number of T_RUNNING processes

4.6.3.3 count_running_priority()

```
int count_running_priority (
            PCB * head,
            int prio )
```

count number of T_RUNNING processes with desired priority `prio` in a circular linked list

Parameters

head	pointer to the head of circular linked list
prio	desired priority (-1, 0, or 1)

Returns

number relevant processes

4.6.3.4 createPCB()

```
PCB* createPCB (
            PCB * Parent )
```

create a PCB

Parameters

parent	the parent process PCB

Returns

the created PCB

creates PCB if parent `Parent` isn't NULL, parent_pid and priority will be inherited instantiated PCB will also be added to `Parent` 's array of children

Parameters

Parent	of parent (if it exists) for the newly created PCB.

Returns

returns the newly created PCB.

4.6.3.5 findPCBByContext()

```
PCB* findPCBByContext (
            ucontext_t * context )
```

find a PCB by `ucontext` in the global PCB list

Parameters

context	the process ucontext to find

Returns

the `PCB`, or `NULL` if it was not found

finds PCB with desired ucontext_t `context`

Parameters

context	the pointer of the given context

Returns

PCB with desired context, if it exists

4.6.3.6 findPCBByPID()

```
PCB* findPCBByPID (
            pid_t pid )
```

find a PCB by pid in the global PCB list

Parameters

pid	the process pid to find

Returns

the PCB, or NULL if it was not found

finds PCB with desired pid `pid`

Parameters

pid	the pid of the process we want to find

Returns

PCB with desired pid, if it exists

4.6.3.7 getLength()

```
int getLength (
            PCB * head )
```

gets length of circular linked list

Parameters

head	pointer to the head of circular linked list

Returns

the length of the pcb list

4.6.3.8 k_free()

```
void k_free (
            PCB * process )
```

free memory of a PCB

Parameters

pcb	the pcb

Returns

none

frees inputted PCB `process`

Parameters

process	pointer to PCB to be freed

Returns

none

4.6.3.9 removePCBFromList()

```
void removePCBFromList (
            PCB ** head,
            PCB * pcb )
```

remove a process from the PCBList

Parameters

list	the `PCBList`
pcb	the process `PCB` to remove

Returns

none

removes a given a PCB `pcb` from a given list

Parameters

head	pointer to head of circular linked list
pcb	the pcb we want to remove

Returns

None

4.6.4 Variable Documentation

4.6.4.1 next_pid

```
pid_t next_pid  [extern]
```

4.6.4.2 pcb_list

```
PCB* pcb_list  [extern]
```

4.7 src/kernel/puser-functions.c File Reference

```
#include "kernel-functions.h"
#include "scheduler.h"
#include "../filesystem/filesystem.h"
#include "../logger/logger.h"
#include "../util/globals.h"
#include "../util/p-errno.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <ucontext.h>
#include <valgrind/valgrind.h>
```

Include dependency graph for puser-functions.c:

Functions

- int p_spawn (void(*func)(), char *argv[], int fd0, int fd1)
- pid_t p_waitpid (pid_t pid, int *wstatus, bool nohang)
- int p_kill (pid_t pid, int sig)
- int p_nice (pid_t pid, int priority)
- void p_sleep (unsigned int time)
- void p_exit (void)
- bool W_WIFEXITED (int status)
- bool W_WIFSTOPPED (int status)
- bool W_WIFCONTINUED (int status)
- bool W_WIFSIGNALED (int status)

Variables

- PCB * current_pcb = NULL
- int ticks = 0

4.7.1 Function Documentation

4.7.1.1 p_exit()

```
void p_exit (
            void  )
```

exits current PCB unconditionally

Returns

none

4.7.1.2 p_kill()

```
int p_kill (
            pid_t pid,
            int sig )
```

sends signal `sig` to PCB with pid `pid`

Parameters

pid	pid of PCB to send signal to
sig	signal to send

Returns

0 on sucess; -1 on failure

4.7.1.3 p_nice()

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

changes priority of PCB with pid `pid` to inputted priority `priority`

Parameters

pid	pid of PCB to change priority of
priority	priority to change to

Returns

0 on sucess; -1 on failure

4.7.1.4 p_sleep()

```
void p_sleep (
            unsigned int time )
```

blocks current PCB for `time` ticks

Parameters

time	ticks to block for

Returns

none

4.7.1.5 p_spawn()

```
int p_spawn (
            void(*)() func,
            char * argv[],
            int fd0,
            int fd1 )
```

spawns PCB with start function `func` , input arguments `argv` , and I/O `fd0` / `fd1`

Parameters

func	start function of PCB
argv	arguments of func
fd0	F_STDIN file descriptor
fd1	F_STDOUT file descriptor

Returns

pid of spawned PCB on success; -1 on failure

4.7.1.6 p_waitpid()

```
pid_t p_waitpid (
            pid_t pid,
```

```
            int * wstatus,
            bool nohang )
```

if nohang is false, waits until relevant PCB(s) changes state if nohang is true, returns immediately

Parameters

pid	pid of PCB to wait for; if -1, wait for any child of current PCB
wstatus	pointer to integer to store status in
nohang	true to return immediately, false to block parent PCB

Returns

0 on success, -1 on failure

4.7.1.7 W_WIFCONTINUED()

```
bool W_WIFCONTINUED (
            int status )
```

Returns

true if the child was continued by a signal

4.7.1.8 W_WIFEXITED()

```
bool W_WIFEXITED (
            int status )
```

Returns

true if the child terminated normally, that is, by a call to p_exit or by returning

4.7.1.9 W_WIFSIGNALED()

```
bool W_WIFSIGNALED (
            int status )
```

Returns

true if the child was terminated by a signal

4.7.1.10 W_WIFSTOPPED()

```
bool W_WIFSTOPPED (
            int status )
```

Returns

true if the child was stopped by a signal

4.7.2 Variable Documentation

4.7.2.1 current_pcb

```
PCB* current_pcb = NULL
```

4.7.2.2 ticks

```
int ticks = 0
```

4.8 src/kernel/puser-functions.h File Reference

```
#include "kernel-functions.h"
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <ucontext.h>
```

Include dependency graph for puser-functions.h: This graph shows which files directly or indirectly include this file:

Functions

- int p_spawn (void(*func)(), char *argv[], int fd0, int fd1)
- pid_t p_waitpid (pid_t pid, int *wstatus, bool nohang)
- int p_kill (pid_t pid, int sig)
- int p_nice (pid_t pid, int priority)
- void p_sleep (unsigned int time)
- void p_exit (void)
- bool W_WIFEXITED (int status)
- bool W_WIFSTOPPED (int status)
- bool W_WIFCONTINUED (int status)
- bool W_WIFSIGNALED (int status)

Variables

- PCB * current_pcb
- int ticks

4.8.1 Function Documentation

4.8.1.1 p_exit()

```
void p_exit (
             void  )
```

exits current PCB unconditionally

Returns

none

4.8.1.2 p_kill()

```
int p_kill (
             pid_t pid,
             int sig )
```

sends signal `sig` to PCB with pid `pid`

Parameters

pid	pid of PCB to send signal to
sig	signal to send

Returns

0 on sucess; -1 on failure

4.8.1.3 p_nice()

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

changes priority of PCB with pid `pid` to inputted priority `priority`

Parameters

pid	pid of PCB to change priority of
priority	priority to change to

Returns

0 on sucess; -1 on failure

4.8.1.4 p_sleep()

```
void p_sleep (
            unsigned int time )
```

blocks current PCB for `time` ticks

Parameters

time	ticks to block for

Returns

none

4.8.1.5 p_spawn()

```
int p_spawn (
            void(*)() func,
            char * argv[],
            int fd0,
            int fd1 )
```

spawns PCB with start function `func` , input arguments `argv` , and I/O `fd0` / `fd1`

Parameters

func	start function of PCB
argv	arguments of func
fd0	F_STDIN file descriptor
fd1	F_STDOUT file descriptor

Returns

pid of spawned PCB on success; -1 on failure

4.8.1.6 p_waitpid()

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

if `nohang` is false, waits until relevant PCB(s) changes state if `nohang` is true, returns immediately

Parameters

pid	pid of PCB to wait for; if -1, wait for any child of current PCB
wstatus	pointer to integer to store status in
nohang	true to return immediately, false to block parent PCB

Returns

0 on success, -1 on failure

4.8.1.7 W_WIFCONTINUED()

```
bool W_WIFCONTINUED (
            int status )
```

Returns

true if the child was continued by a signal

4.8.1.8 W_WIFEXITED()

```
bool W_WIFEXITED (
            int status )
```

Returns

true if the child terminated normally, that is, by a call to p_exit or by returning

4.8.1.9 W_WIFSIGNALED()

```
bool W_WIFSIGNALED (
            int status )
```

Returns

true if the child was terminated by a signal

4.8.1.10 W_WIFSTOPPED()

```
bool W_WIFSTOPPED (
            int status )
```

Returns

true if the child was stopped by a signal

4.8.2 Variable Documentation

4.8.2.1 current_pcb

```
PCB* current_pcb  [extern]
```

4.8.2.2 ticks

```
int ticks  [extern]
```

4.9 src/kernel/scheduler.c File Reference

```
#include "puser-functions.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <ucontext.h>
#include <sys/time.h>
#include "../util/globals.h"
#include "../filesystem/filesystem.h"
#include "../logger/logger.h"
#include <time.h>
#include <valgrind/valgrind.h>
```

Include dependency graph for scheduler.c:

Functions

- static void scheduler (void)
- static void reaper ()
- static void alarmHandler (int signum)
- static void setAlarmHandler (void)
- static void setTimer (void)
- static void freeStacks (void)
- void start_scheduler ()

Variables

- static ucontext_t mainContext
- ucontext_t schedulerContext
- ucontext_t reaperContext
- static ucontext_t * activeContext = NULL
- static const int centisecond = 10000

4.9.1 Function Documentation

4.9.1.1 alarmHandler()

```
static void alarmHandler (
            int signum )  [static]
```

alarm handler that is invoked when alarm is signalled at every quanta increments ticks and sets current context to scheduler, effectively allowing the scheduler to run at every quanta

Returns

none

4.9.1.2 freeStacks()

```
static void freeStacks (
            void  )  [static]
```

frees all contexts prior to exit

Returns

none

4.9.1.3 reaper()

```
static void reaper ( )  [static]
```

reaper function that runs at termination of PCB increments ticks and sets current context back to scheduler

Returns

none

4.9.1.4 scheduler()

```
static void scheduler (
            void  )  [static]
```

scheduler function - function to be run at every tick decides which PCB to be run for the remainder of current tick

Returns

none

4.9.1.5 setAlarmHandler()

```
static void setAlarmHandler (
            void  )  [static]
```

sets alarmHandler to be called when alarm is signalled

Returns

none

4.9.1.6 setTimer()

```
static void setTimer (
            void  )  [static]
```

sets timer to invoke alarm every centisecond

Returns

none

4.9.1.7 start_scheduler()

```
void start_scheduler ( )
```

initializes and start scheduler

Returns

none

4.9.2 Variable Documentation

4.9.2.1 activeContext

```
ucontext_t* activeContext = NULL  [static]
```

4.9.2.2 centisecond

```
const int centisecond = 10000  [static]
```

4.9.2.3 mainContext

```
ucontext_t mainContext  [static]
```

4.9.2.4 reaperContext

```
ucontext_t reaperContext
```

4.9.2.5 schedulerContext

```
ucontext_t schedulerContext
```

4.10 src/kernel/scheduler.h File Reference

```
#include "puser-functions.h"
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <ucontext.h>
#include <sys/time.h>
```

Include dependency graph for scheduler.h: This graph shows which files directly or indirectly include this file:

Functions

- void init_scheduler ()
- void start_scheduler ()
- void setAlarmHandler ()
- void setTimer ()

Variables

- ucontext_t schedulerContext
- ucontext_t reaperContext

4.10.1 Function Documentation

4.10.1.1 init_scheduler()

```
void init_scheduler ( )
```

4.10.1.2 setAlarmHandler()

```
void setAlarmHandler ( )
```

4.10.1.3 setTimer()

```
void setTimer ( )
```

4.10.1.4 start_scheduler()

```
void start_scheduler ( )
```

initializes and start scheduler

Returns

none

4.10.2 Variable Documentation

4.10.2.1 reaperContext

```
ucontext_t reaperContext  [extern]
```

4.10.2.2 schedulerContext

```
ucontext_t schedulerContext  [extern]
```

4.11 src/logger/logger.c File Reference

```
#include "logger.h"
#include "stdio.h"
#include "../kernel/puser-functions.h"
```

Include dependency graph for logger.c:

Functions

- void log_schedule_event (int pid, int prio, char *process_name)
  Logs a scheduling event.
- void log_create_event (int pid, int prio, char *process_name)
  Logs a process creation event.
- void log_signaled_event (int pid, int prio, char *process_name)
  Logs a signalling event.
- void log_exited_event (int pid, int prio, char *process_name)
  Logs an exiting event (natural termination via p_exit) event.
- void log_zombie_event (int pid, int prio, char *process_name)
  Logs zombie event.
- void log_orphan_event (int pid, int prio, char *process_name)
  Logs orphan event.
- void log_waited_event (int pid, int prio, char *process_name)

Logs waiting event.

- void log_nice_event (int pid, int old_prio, int new_prio, char ∗process_name)

  Logs a priority change event.
- void log_blocked_event (int pid, int prio, char ∗process_name)

  Logs a blocked event (via waitpid).
- void log_unblocked_event (int pid, int prio, char ∗process_name)

  Logs an unblocked event.
- void log_stopped_event (int pid, int prio, char ∗process_name)

  Logs a stopped event (via signalling).
- void log_continued_event (int pid, int prio, char ∗process_name)

  Logs a continued event.

Variables

- FILE ∗ logfile

4.11.1 Function Documentation

4.11.1.1 log_blocked_event()

```
void log_blocked_event (
            int pid,
            int prio,
            char * process_name )
```

Logs a blocked event (via waitpid).

Parameters

pid	Process ID of the blocked process.
prio	Priority of the process.
process_name	Name of the process.

4.11.1.2 log_continued_event()

```
void log_continued_event (
            int pid,
            int prio,
            char * process_name )
```

Logs a continued event.

Parameters

pid	Process ID of the continued process.
prio	Priority of the process.
process_name	Name of the process.

4.11.1.3 log_create_event()

```
void log_create_event (
            int pid,
            int prio,
            char * process_name )
```

Logs a process creation event.

Parameters

pid	Process ID of the created process.
prio	Priority of the created process.
process_name	Name of the created process.

4.11.1.4 log_exited_event()

```
void log_exited_event (
            int pid,
            int prio,
            char * process_name )
```

Logs an exiting event (natural termination via p_exit) event.

Parameters

pid	Process ID of the exited process.
prio	Priority of the exited process.
process_name	Name of the exited process.

4.11.1.5 log_nice_event()

```
void log_nice_event (
            int pid,
            int old_prio,
```

```
            int new_prio,
            char * process_name )
```

Logs a priority change event.

Parameters

pid	Process ID of the changed process.
old_prio	Old priority of the process.
new_prio	New priority of the process.
process_name	Name of the process.

4.11.1.6 log_orphan_event()

```
void log_orphan_event (
            int pid,
            int prio,
            char * process_name )
```

Logs orphan event.

Parameters

pid	Process ID of the ophaned process.
prio	Priority of the ophaned process.
process_name	Name of the ophaned process.

4.11.1.7 log_schedule_event()

```
void log_schedule_event (
            int pid,
            int prio,
            char * process_name )
```

Logs a scheduling event.

Parameters

pid	Process ID of the scheduled process.
prio	Priority of the scheduled process.
process_name	Name of the scheduled process.

4.11.1.8 log_signaled_event()

```
void log_signaled_event (
            int pid,
            int prio,
            char * process_name )
```

Logs a signalling event.

Parameters

pid	Process ID of the signalled process.
prio	Priority of the signalled process.
process_name	Name of the signalled process.

4.11.1.9 log_stopped_event()

```
void log_stopped_event (
            int pid,
            int prio,
            char * process_name )
```

Logs a stopped event (via signalling).

Parameters

pid	Process ID of the stopped process.
prio	Priority of the process.
process_name	Name of the process.

4.11.1.10 log_unblocked_event()

```
void log_unblocked_event (
            int pid,
            int prio,
            char * process_name )
```

Logs an unblocked event.

Parameters

pid	Process ID of the unblocked process.
prio	Priority of the process.
process_name	Name of the process.

4.11.1.11 log_waited_event()

```
void log_waited_event (
            int pid,
            int prio,
            char * process_name )
```

Logs waiting event.

Parameters

pid	Process ID of the waiting process.
prio	Priority of the waiting process.
process_name	Name of the waiting process.

4.11.1.12 log_zombie_event()

```
void log_zombie_event (
            int pid,
            int prio,
            char * process_name )
```

Logs zombie event.

Parameters

pid	Process ID of the zombied process.
prio	Priority of the zombied process.
process_name	Name of the zombied process.

4.11.2 Variable Documentation

4.11.2.1 logfile

```
FILE* logfile
```

4.12 src/logger/logger.h File Reference

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

Include dependency graph for logger.h: This graph shows which files directly or indirectly include this file:

Functions

- void log_schedule_event (int pid, int prio, char ∗process_name)

  Logs a scheduling event.
- void log_create_event (int pid, int prio, char ∗process_name)

  Logs a process creation event.
- void log_signaled_event (int pid, int prio, char ∗process_name)

  Logs a signalling event.
- void log_exited_event (int pid, int prio, char ∗process_name)

  Logs an exiting event (natural termination via p_exit) event.
- void log_zombie_event (int pid, int prio, char ∗process_name)

  Logs zombie event.
- void log_orphan_event (int pid, int prio, char ∗process_name)

  Logs orphan event.
- void log_waited_event (int pid, int prio, char ∗process_name)

  Logs waiting event.
- void log_nice_event (int pid, int old_prio, int new_prio, char ∗process_name)

  Logs a priority change event.
- void log_blocked_event (int pid, int prio, char ∗process_name)

  Logs a blocked event (via waitpid).
- void log_unblocked_event (int pid, int prio, char ∗process_name)

  Logs an unblocked event.
- void log_stopped_event (int pid, int prio, char ∗process_name)

  Logs a stopped event (via signalling).
- void log_continued_event (int pid, int prio, char ∗process_name)

  Logs a continued event.

Variables

- FILE ∗ logfile

4.12.1 Function Documentation

4.12.1.1 log_blocked_event()

```
void log_blocked_event (
            int pid,
            int prio,
            char * process_name )
```

Logs a blocked event (via waitpid).

Parameters

pid	Process ID of the blocked process.
prio	Priority of the process.
process_name	Name of the process.

4.12.1.2 log_continued_event()

```
void log_continued_event (
            int pid,
            int prio,
            char * process_name )
```

Logs a continued event.

Parameters

pid	Process ID of the continued process.
prio	Priority of the process.
process_name	Name of the process.

4.12.1.3 log_create_event()

```
void log_create_event (
            int pid,
            int prio,
            char * process_name )
```

Logs a process creation event.

Parameters

pid	Process ID of the created process.
prio	Priority of the created process.
process_name	Name of the created process.

4.12.1.4 log_exited_event()

```
void log_exited_event (
            int pid,
            int prio,
            char * process_name )
```

Logs an exiting event (natural termination via p_exit) event.

Parameters

pid	Process ID of the exited process.
prio	Priority of the exited process.
process_name	Name of the exited process.

4.12.1.5 log_nice_event()

```
void log_nice_event (
            int pid,
            int old_prio,
            int new_prio,
            char * process_name )
```

Logs a priority change event.

Parameters

pid	Process ID of the changed process.
old_prio	Old priority of the process.
new_prio	New priority of the process.
process_name	Name of the process.

4.12.1.6 log_orphan_event()

```
void log_orphan_event (
            int pid,
            int prio,
            char * process_name )
```

Logs orphan event.

Parameters

pid	Process ID of the ophaned process.
prio	Priority of the ophaned process.
process_name	Name of the ophaned process.

4.12.1.7 log_schedule_event()

```
void log_schedule_event (
            int pid,
            int prio,
            char * process_name )
```

Logs a scheduling event.

Parameters

pid	Process ID of the scheduled process.
prio	Priority of the scheduled process.
process_name	Name of the scheduled process.

4.12.1.8 log_signaled_event()

```
void log_signaled_event (
            int pid,
            int prio,
            char * process_name )
```

Logs a signalling event.

Parameters

pid	Process ID of the signalled process.
prio	Priority of the signalled process.
process_name	Name of the signalled process.

4.12.1.9 log_stopped_event()

```
void log_stopped_event (
            int pid,
            int prio,
            char * process_name )
```

Logs a stopped event (via signalling).

Parameters

pid	Process ID of the stopped process.
prio	Priority of the process.
process_name	Name of the process.

4.12.1.10 log_unblocked_event()

```
void log_unblocked_event (
            int pid,
```

```
            int prio,
            char * process_name )
```

Logs an unblocked event.

Parameters

pid	Process ID of the unblocked process.
prio	Priority of the process.
process_name	Name of the process.

4.12.1.11 log_waited_event()

```
void log_waited_event (
            int pid,
            int prio,
            char * process_name )
```

Logs waiting event.

Parameters

pid	Process ID of the waiting process.
prio	Priority of the waiting process.
process_name	Name of the waiting process.

4.12.1.12 log_zombie_event()

```
void log_zombie_event (
            int pid,
            int prio,
            char * process_name )
```

Logs zombie event.

Parameters

pid	Process ID of the zombied process.
prio	Priority of the zombied process.
process_name	Name of the zombied process.

4.12.2 Variable Documentation

4.12.2.1 logfile

```
FILE* logfile  [extern]
```

4.13 src/pennfat/fat.c File Reference

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <time.h>
#include "fat.h"
#include "safe.h"
```

Include dependency graph for fat.c:

Functions

- int mem_idx (uint16_t *fat, int block_idx)
- int get_free_block (uint16_t *fat)
- void delete_chain (uint16_t *fat, int head)
- void build_chain (uint16_t *fat, int fs_fd, int curr_block, char *data, int n_bytes)
- int fill_chain (uint16_t *fat, int fs_fd, int head, int chain_size, char *buffer, int buffer_size)
- void add_file (uint16_t *fat, int fs_fd, int dir_head, const char *filename)
- void write_file (uint16_t *fat, int fs_fd, point_t location, dir_entry_t entry)
- void read_chain (uint16_t *fat, int fs_fd, int head, char *buffer, int chain_bytes)
- bool find_file (uint16_t *fat, int fs_fd, int dir_head, const char *filename, point_t *loc, dir_entry_t *ret)
- bool valid_filename (char *str)
- void fs_getmeta (uint16_t *fat, int fs_fd, int *n_blocks, int *block_size)
- int fs_mount (char *fs_name, uint16_t **fat)
- void fs_unmount (uint16_t **fat, int fs_fd)
- bool fs_touch (uint16_t *fat, int fs_fd, const char *target)
- bool fs_mv (uint16_t *fat, int fs_fd, const char *old_name, const char *new_name)
- bool fs_mark_deleted (uint16_t *fat, int fs_fd, const char *target)
- bool fs_rm (uint16_t *fat, int fs_fd, const char *target)
- char * fs_cat (uint16_t *fat, int fs_fd, int input_mode, int output_mode, char *input_str, char *input_files[], char *output_file)
- bool fs_cp (uint16_t *fat, int fs_fd, const char *source, const char *dest)
- bool fs_cp_mode (uint16_t *fat, int fs_fd, const char *source, const char *dest, bool host_in, bool host_out)
- void fs_ls_single (dir_entry_t *entry)
- void fs_ls (uint16_t *fat, int fs_fd)
- uint8_t fs_chmod (uint16_t *fat, int fs_fd, const char *target, uint8_t permissions)

Variables

- const int DIR_ENTRY_SIZE = 64
- const int ROOTDIR = 1
- const int DEFAULT_PERMISSIONS = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH
- const int LASTBLOCK = 0xFFFF
- const int BITS_PER_BYTE = 8
- const int BYTE_SIZE = 1 << BITS_PER_BYTE
- const int FILETYPE_UNKNOWN = 0
- const int FILETYPE_FILE = 1

- const int FILETYPE_DIRECTORY = 2
- const int FILETYPE_LINK = 4
- const int FILENAME_ENDDIR = 0
- const int FILENAME_DEL_UNUSED = 1
- const int FILENAME_DEL_INUSE = 2
- const int FILEPERM_NONE = 0
- const int FILEPERM_RD = 0b100
- const int FILEPERM_WR = 0b010
- const int FILEPERM_EX = 0b001

4.13.1 Function Documentation

4.13.1.1 add_file()

```
void add_file (
            uint16_t * fat,
            int fs_fd,
            int dir_head,
            const char * filename )
```

add a new empty file to the directory, allocating new blocks as necessary

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
dirhead	the first index of the directory chain (`ROOTDIR` or `1` for the root dir)
filename	the file to add

Returns

none

4.13.1.2 build_chain()

```
void build_chain (
            uint16_t * fat,
            int fs_fd,
            int curr_block,
            char * data,
            int n_bytes )
```

allocates data as a FAT chain

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
curr_block	first index of the chain to build
data	what to copy to memory
n_bytes	length of `data`

Returns

none

4.13.1.3 delete_chain()

```
void delete_chain (
            uint16_t * fat,
            int head )
```

traverse down the fat chain, marking them all as deleted

Parameters

fat	filesystem
head	the first index of the chain

Returns

none

4.13.1.4 fill_chain()

```
int fill_chain (
            uint16_t * fat,
            int fs_fd,
            int head,
            int chain_size,
            char * buffer,
            int buffer_size )
```

fill a FAT chain with data

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
head	the first index of the chain
buffer	what to read from
buffer_size	length of `buffer`

Returns

the number of bytes written

4.13.1.5 find_file()

```
bool find_file (
            uint16_t * fat,
            int fs_fd,
            int dir_head,
            const char * filename,
            point_t * loc,
            dir_entry_t * ret )
```

Finds a file or directory in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
dir_head	Index of the first block in the directory.
filename	Name of the file or directory to find.
loc	Pointer to a point_t structure to store the location (block index and entry index).
ret	Pointer to a dir_entry_t structure to store the found directory entry.

Returns

Returns true if the file or directory is found; otherwise, false.

4.13.1.6 fs_cat()

```
char* fs_cat (
            uint16_t * fat,
            int fs_fd,
            int input_mode,
            int output_mode,
            char * input_str,
            char * input_files[],
            char * output_file )
```

Concatenates input strings or files and outputs the result to the terminal or a file.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
input_mode	Mode for input: 0 for input string, >0 for input files.
output_mode	Mode for output: 0 for stdout, 1 for file overwrite, 2 for file append.
input_str	Input string to concatenate (used when input_mode is 0).
input_files	Array of input file names (used when input_mode is >0).
output_file	Name of the output file (used when output_mode is 1 or 2).

Returns

Returns the concatenated output string if output_mode is 0; otherwise, returns NULL.

4.13.1.7 fs_chmod()

```
uint8_t fs_chmod (
            uint16_t * fat,
            int fs_fd,
            const char * target,
            uint8_t permissions )
```

Changes the permissions of a file or directory in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
target	Name of the file or directory to change permissions.
permissions	New permissions to set.

Returns

Returns the old permissions before the change.

4.13.1.8 fs_cp()

```
bool fs_cp (
            uint16_t * fat,
            int fs_fd,
            const char * source,
            const char * dest )
```

Copies a file or directory to a destination in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
source	Name of the source file or directory to copy.
dest	Name of the destination file or directory.

Returns

Returns true if the copy is successful; otherwise, false.

4.13.1.9 fs_cp_mode()

```
bool fs_cp_mode (
            uint16_t * fat,
            int fs_fd,
            const char * source,
            const char * dest,
            bool host_in,
            bool host_out )
```

Copies a file or directory with specified input and output modes in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
source	Name of the source file or directory to copy.
dest	Name of the destination file or directory.
host_in	Input mode: true for hostOS to PennFAT, false for PennFAT to PennFAT.
host_out	Output mode: true for PennFAT to hostOS, false for PennFAT to PennFAT.

Returns

Returns true if the copy is successful; otherwise, false.

4.13.1.10 fs_getmeta()

```
void fs_getmeta (
            uint16_t * fat,
            int fs_fd,
            int * n_blocks,
            int * block_size )
```

Retrieves metadata information from the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
n_blocks	Pointer to an integer to store the number of blocks in the filesystem.
block_size	Pointer to an integer to store the block size in bytes.

Returns

None.

4.13.1.11 fs_ls()

```
void fs_ls (
            uint16_t * fat,
            int fs_fd )
```

Displays information about all directory entries in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.

Returns

None.

4.13.1.12 fs_ls_single()

```
void fs_ls_single (
            dir_entry_t * entry )
```

Displays information about a single directory entry.

Parameters

entry	Pointer to the directory entry.

Returns

None.

4.13.1.13 fs_mark_deleted()

```
bool fs_mark_deleted (
            uint16_t * fat,
            int fs_fd,
            const char * target )
```

Marks a file or directory as deleted in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
target	Name of the file or directory to mark as deleted.

Returns

Returns true if the file or directory is successfully marked as deleted; otherwise, false.

4.13.1.14 fs_mount()

```
int fs_mount (
             char * fs_name,
             uint16_t ** fat )
```

Mounts a file system.

Parameters

fs_name	The name of the file system to mount.
fat	Pointer to FAT.

Returns

Returns the file descriptor of the mounted file system on success. On failure, returns -1.

4.13.1.15 fs_mv()

```
bool fs_mv (
             uint16_t * fat,
             int fs_fd,
             const char * old_name,
             const char * new_name )
```

Moves or renames a file in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
old_name	Name of the file to move or rename.
new_name	New name or path for the file.

Returns

Returns true if the file is successfully moved or renamed; otherwise, false.

4.13.1.16 fs_rm()

```
bool fs_rm (
            uint16_t * fat,
            int fs_fd,
            const char * target )
```

Removes (deletes) a file or directory from the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
target	Name of the file or directory to remove.

Returns

Returns true if the file or directory is successfully removed; otherwise, false.

4.13.1.17 fs_touch()

```
bool fs_touch (
            uint16_t * fat,
            int fs_fd,
            const char * target )
```

Creates or updates a file in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
target	Name of the file to create or update.

Returns

Returns true if a new file is created; otherwise, false if an existing file is updated.

4.13.1.18 fs_unmount()

```
void fs_unmount (
            uint16_t ** fat,
            int fs_fd )
```

Unmounts a file system.

Parameters

fat	Pointer to FAT.
fs↩ _fd	File descriptor of the filesystem.

Returns

None.

4.13.1.19 get_free_block()

```
int get_free_block (
            uint16_t * fat )
```

search for an open block (where value is `0`)

Parameters

fat	filesystem

Returns

the block index on success, `0` on failure

4.13.1.20 mem_idx()

```
int mem_idx (
            uint16_t * fat,
            int block_idx )
```

get the address in memory of the specified block index

Parameters

fat	filesystem
block_idx	block index

Returns

the address/index in memory of `block_idx`

4.13.1.21 read_chain()

```
void read_chain (
            uint16_t * fat,
            int fs_fd,
            int head,
            char * buffer,
            int chain_bytes )
```

Reads a FAT chain from the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
head	Index of the first block in the chain.
buffer	Buffer to store the read data.
chain_bytes	Total number of bytes to read from the chain.

Returns

None.

4.13.1.22 valid_filename()

```
bool valid_filename (
            char * str )
```

Checks if a string is a valid filename.

Parameters

str	The string to be checked.

Returns

Returns true if the string is a valid filename; otherwise, false.

4.13.1.23 write_file()

```
void write_file (
            uint16_t * fat,
            int fs_fd,
            point_t location,
            dir_entry_t entry )
```

seek and write a file block to memory

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
location	point in memory
entry	the file block

Returns

none

4.13.2 Variable Documentation

4.13.2.1 BITS_PER_BYTE

```
const int BITS_PER_BYTE = 8
```

4.13.2.2 BYTE_SIZE

```
const int BYTE_SIZE = 1 << BITS_PER_BYTE
```

4.13.2.3 DEFAULT_PERMISSIONS

```
const int DEFAULT_PERMISSIONS = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH
```

4.13.2.4 DIR_ENTRY_SIZE

```
const int DIR_ENTRY_SIZE = 64
```

4.13.2.5 FILENAME_DEL_INUSE

```
const int FILENAME_DEL_INUSE = 2
```

4.13.2.6 FILENAME_DEL_UNUSED

```
const int FILENAME_DEL_UNUSED = 1
```

4.13.2.7 FILENAME_ENDDIR

```
const int FILENAME_ENDDIR = 0
```

4.13.2.8 FILEPERM_EX

```
const int FILEPERM_EX = 0b001
```

4.13.2.9 FILEPERM_NONE

```
const int FILEPERM_NONE = 0
```

4.13.2.10 FILEPERM_RD

```
const int FILEPERM_RD = 0b100
```

4.13.2.11 FILEPERM_WR

```
const int FILEPERM_WR = 0b010
```

4.13.2.12 FILETYPE_DIRECTORY

```
const int FILETYPE_DIRECTORY = 2
```

4.13.2.13 FILETYPE_FILE

```
const int FILETYPE_FILE = 1
```

4.13.2.14 FILETYPE_LINK

```
const int FILETYPE_LINK = 4
```

4.13.2.15 FILETYPE_UNKNOWN

```
const int FILETYPE_UNKNOWN = 0
```

4.13.2.16 LASTBLOCK

```
const int LASTBLOCK = 0xFFFF
```

4.13.2.17 ROOTDIR

```
const int ROOTDIR = 1
```

4.14 src/pennfat/fat.h File Reference

```
#include <stdbool.h>
```
Include dependency graph for fat.h: This graph shows which files directly or indirectly include this file:

Data Structures

- struct directory_entry
- struct point

Macros

- #define FAT_BLOCKS(x) ((x) >> BITS_PER_BYTE)
- #define BLOCK_SIZE(x) (BYTE_SIZE << ((x) & 0xFF))

Typedefs

- typedef struct directory_entry dir_entry_t
- typedef struct point point_t

Functions

- void read_chain (uint16_t *fat, int fs_fd, int head, char *buffer, int chain_bytes)
- bool find_file (uint16_t *fat, int fs_fd, int dir_head, const char *filename, point_t *loc, dir_entry_t *ret)
- bool valid_filename (char *str)
- void fs_getmeta (uint16_t *fat, int fs_fd, int *n_blocks, int *block_size)
- int fs_mount (char *fs_name, uint16_t **fat)
- void fs_unmount (uint16_t **fat, int fs_fd)
- bool fs_touch (uint16_t *fat, int fs_fd, const char *target)
- bool fs_mv (uint16_t *fat, int fs_fd, const char *old_name, const char *new_name)
- bool fs_rm (uint16_t *fat, int fs_fd, const char *target)
- bool fs_mark_deleted (uint16_t *fat, int fs_fd, const char *target)
- char * fs_cat (uint16_t *fat, int fs_fd, int input_mode, int output_mode, char *input_str, char *input_files[], char *output_file)
- bool fs_cp (uint16_t *fat, int fs_fd, const char *source, const char *dest)
- bool fs_cp_mode (uint16_t *fat, int fs_fd, const char *source, const char *dest, bool host_in, bool host_out)
- void fs_ls_single (dir_entry_t *entry)
- void fs_ls (uint16_t *fat, int fs_fd)
- uint8_t fs_chmod (uint16_t *fat, int fs_fd, const char *target, uint8_t permissions)

Variables

- const int DIR_ENTRY_SIZE
- const int ROOTDIR
- const int DEFAULT_PERMISSIONS
- const int LASTBLOCK
- const int BITS_PER_BYTE
- const int BYTE_SIZE
- const int FILETYPE_UNKNOWN
- const int FILETYPE_FILE
- const int FILETYPE_DIRECTORY
- const int FILETYPE_LINK
- const int FILENAME_ENDDIR
- const int FILENAME_DEL_UNUSED
- const int FILENAME_DEL_INUSE
- const int FILEPERM_NONE
- const int FILEPERM_RD
- const int FILEPERM_WR
- const int FILEPERM_EX

4.14.1 Macro Definition Documentation

4.14.1.1 BLOCK_SIZE

```
#define BLOCK_SIZE(
              x ) (BYTE_SIZE << ((x) & 0xFF))
```

4.14.1.2 FAT_BLOCKS

```
#define FAT_BLOCKS(
              x ) ((x) >> BITS_PER_BYTE)
```

4.14.2 Typedef Documentation

4.14.2.1 dir_entry_t

```
typedef struct directory_entry dir_entry_t
```

4.14.2.2 point_t

```
typedef struct point point_t
```

4.14.3 Function Documentation

4.14.3.1 find_file()

```
bool find_file (
             uint16_t * fat,
             int fs_fd,
             int dir_head,
             const char * filename,
             point_t * loc,
             dir_entry_t * ret )
```

search for a filename in the directory use `NULL` for `loc` and `ret` to simply check if the file exists

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
dir_head	the first index of the directory chain (`ROOTDIR` or `1` for the root dir)
filename	the file to search for
loc	will be set to the block (`loc.first`) & entry number (`loc.second`) of the file, if the file is found
ret	the directory entry of the file, if the file is found

Returns

`true` if the file is found, `false` otherwise

Finds a file or directory in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
dir_head	Index of the first block in the directory.
filename	Name of the file or directory to find.
loc	Pointer to a point_t structure to store the location (block index and entry index).
ret	Pointer to a dir_entry_t structure to store the found directory entry.

Returns

Returns true if the file or directory is found; otherwise, false.

4.14.3.2 fs_cat()

```
char* fs_cat (
            uint16_t * fat,
            int fs_fd,
            int input_mode,
            int output_mode,
            char * input_str,
            char * input_files[],
            char * output_file )
```

cat a number of files or the `input_str`, output to buffer or another file

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
input_mode	`0` for `stdin`, otherwise the number of files in `input_files`
output_mode	`0` for `stdout`, `1` for overwrite, `2` for append
input_str	if `input_mode` is `0`, the input
input_files	if `input_mode` is not `0`, the files to concatenate
output_file	if `output_mode` is not `0`, the file to output to

Returns

if `output_mode` is `0`, the output buffer

Concatenates input strings or files and outputs the result to the terminal or a file.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
input_mode	Mode for input: 0 for input string, >0 for input files.
output_mode	Mode for output: 0 for stdout, 1 for file overwrite, 2 for file append.
input_str	Input string to concatenate (used when input_mode is 0).
input_files	Array of input file names (used when input_mode is >0).
output_file	Name of the output file (used when output_mode is 1 or 2).

Returns

Returns the concatenated output string if output_mode is 0; otherwise, returns NULL.

4.14.3.3 fs_chmod()

```
uint8_t fs_chmod (
            uint16_t * fat,
            int fs_fd,
            const char * target,
            uint8_t permissions )
```

change permissions

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
target	the file to change permissions of
permissions	the new permissions

Returns

the old permissions

Changes the permissions of a file or directory in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
target	Name of the file or directory to change permissions.
permissions	New permissions to set.

Returns

Returns the old permissions before the change.

4.14.3.4 fs_cp()

```
bool fs_cp (
            uint16_t * fat,
            int fs_fd,
            const char * source,
            const char * dest )
```

copy a file

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
source	the file to copy from
dest	the file to copy to; created if necessary, or overwritten if not

Returns

`true` on success, `false` if `source` was not found

Copies a file or directory to a destination in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
source	Name of the source file or directory to copy.
dest	Name of the destination file or directory.

Returns

Returns true if the copy is successful; otherwise, false.

4.14.3.5 fs_cp_mode()

```
bool fs_cp_mode (
            uint16_t * fat,
            int fs_fd,
            const char * source,
            const char * dest,
```

```
        bool host_in,
        bool host_out )
```

copy a file (may input/output with host OS)

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
source	the file to copy from
dest	the file to copy to; created if necessary, or overwritten if not
host_in	if `true`, input file from host OS
host_out	if `true`, output file to host OS

Returns

`true` on success, `false` if `source` was not found

Copies a file or directory with specified input and output modes in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
source	Name of the source file or directory to copy.
dest	Name of the destination file or directory.
host_in	Input mode: true for hostOS to PennFAT, false for PennFAT to PennFAT.
host_out	Output mode: true for PennFAT to hostOS, false for PennFAT to PennFAT.

Returns

Returns true if the copy is successful; otherwise, false.

4.14.3.6 fs_getmeta()

```
void fs_getmeta (
            uint16_t * fat,
            int fs_fd,
            int * n_blocks,
            int * block_size )
```

get the metadata of a filesystem

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
n_blocks	set to the number of blocks in FAT
block_size	set to the size of a block

Returns

none

Retrieves metadata information from the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
n_blocks	Pointer to an integer to store the number of blocks in the filesystem.
block_size	Pointer to an integer to store the block size in bytes.

Returns

None.

4.14.3.7 fs_ls()

```
void fs_ls (
            uint16_t * fat,
            int fs_fd )
```

list directory

Parameters

fat	filesystem
fs↩ _fd	filesystem file descriptor

Returns

none

Displays information about all directory entries in the filesystem.

Parameters

fat	Pointer to FAT.
fs↩ _fd	File descriptor of the filesystem.

Returns

None.

4.14.3.8 fs_ls_single()

```
void fs_ls_single (
            dir_entry_t * entry )
```

list information for a single file

Parameters

entry	the directory entry for the file

Returns

none

Displays information about a single directory entry.

Parameters

entry	Pointer to the directory entry.

Returns

None.

4.14.3.9 fs_mark_deleted()

```
bool fs_mark_deleted (
            uint16_t * fat,
            int fs_fd,
            const char * target )
```

mark a file as deleted with `FILENAME_DEL_INUSE`, `-2`; it is still in use by another process, but shouldn't be able to be accessed anew

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
target	the file to mark

Returns

`true` if the file was marked as deleted, `false` otherwise

Marks a file or directory as deleted in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
target	Name of the file or directory to mark as deleted.

Returns

Returns true if the file or directory is successfully marked as deleted; otherwise, false.

4.14.3.10 fs_mount()

```
int fs_mount (
             char * fs_name,
             uint16_t ** fat )
```

mount a filesystem & map fat to memory

Parameters

fs_name	filesystem filename
fat	set to the filesytem

Returns

the filesystem file descriptor (`fs_fd`)

Mounts a file system.

Parameters

fs_name	The name of the file system to mount.
fat	Pointer to FAT.

Returns

Returns the file descriptor of the mounted file system on success. On failure, returns -1.

4.14.3.11 fs_mv()

```
bool fs_mv (
             uint16_t * fat,
             int fs_fd,
             const char * old_name,
             const char * new_name )
```

rename a file

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
old_name	the file to rename
new_name	the new name for `old_name`

Returns

`true` if the file is renamed (and found), `false` otherwise

Moves or renames a file in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
old_name	Name of the file to move or rename.
new_name	New name or path for the file.

Returns

Returns true if the file is successfully moved or renamed; otherwise, false.

4.14.3.12 fs_rm()

```
bool fs_rm (
            uint16_t * fat,
            int fs_fd,
            const char * target )
```

delete a file

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
target	the file to delete

Returns

`true` if the file was deleted, `false` otherwise

Removes (deletes) a file or directory from the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
target	Name of the file or directory to remove.

Returns

Returns true if the file or directory is successfully removed; otherwise, false.

4.14.3.13 fs_touch()

```
bool fs_touch (
            uint16_t * fat,
            int fs_fd,
            const char * target )
```

touch a file; if it exists, update the timestamp, otherwise create the file

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
target	the target filename

Returns

`true` if the file is created, `false` otherwise

Creates or updates a file in the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
target	Name of the file to create or update.

Returns

Returns true if a new file is created; otherwise, false if an existing file is updated.

4.14.3.14 fs_unmount()

```
void fs_unmount (
            uint16_t ** fat,
            int fs_fd )
```

unmount a filesystem

Parameters

fat	pointer to the filesystem; will be unmapped from memory
fs↩ _fd	filesystem file descriptor; will be closed

Returns

none

Unmounts a file system.

Parameters

fat	Pointer to FAT.
fs↩ _fd	File descriptor of the filesystem.

Returns

None.

4.14.3.15 read_chain()

```
void read_chain (
            uint16_t * fat,
            int fs_fd,
            int head,
            char * buffer,
            int chain_bytes )
```

read a FAT chain

Parameters

fat	filesystem
fs_fd	filesystem file descriptor
head	the first index of the chain
buffer	what to read into; assume this has enough space
chain_bytes	length of the chain

Returns

none

Reads a FAT chain from the filesystem.

Parameters

fat	Pointer to FAT.
fs_fd	File descriptor of the filesystem.
head	Index of the first block in the chain.
buffer	Buffer to store the read data.
chain_bytes	Total number of bytes to read from the chain.

Returns

None.

4.14.3.16 valid_filename()

```
bool valid_filename (
            char * str )
```

check whether a filename is valid

Parameters

str	filename

Returns

`true` if `str` is POSIX-valid (consists of [A-Za-z0-9._-]) and 32 bytes null-terminated (31 characters), `false` & print otherwise

Checks if a string is a valid filename.

Parameters

str	The string to be checked.

Returns

Returns true if the string is a valid filename; otherwise, false.

4.14.4 Variable Documentation

4.14.4.1 BITS_PER_BYTE

```
const int BITS_PER_BYTE  [extern]
```

4.14.4.2 BYTE_SIZE

```
const int BYTE_SIZE  [extern]
```

4.14.4.3 DEFAULT_PERMISSIONS

```
const int DEFAULT_PERMISSIONS  [extern]
```

4.14.4.4 DIR_ENTRY_SIZE

```
const int DIR_ENTRY_SIZE  [extern]
```

4.14.4.5 FILENAME_DEL_INUSE

```
const int FILENAME_DEL_INUSE  [extern]
```

4.14.4.6 FILENAME_DEL_UNUSED

```
const int FILENAME_DEL_UNUSED  [extern]
```

4.14.4.7 FILENAME_ENDDIR

```
const int FILENAME_ENDDIR  [extern]
```

4.14.4.8 FILEPERM_EX

```
const int FILEPERM_EX  [extern]
```

4.14.4.9 FILEPERM_NONE

```
const int FILEPERM_NONE  [extern]
```

4.14.4.10 FILEPERM_RD

```
const int FILEPERM_RD  [extern]
```

4.14.4.11 FILEPERM_WR

```
const int FILEPERM_WR  [extern]
```

4.14.4.12 FILETYPE_DIRECTORY

```
const int FILETYPE_DIRECTORY  [extern]
```

4.14.4.13 FILETYPE_FILE

```
const int FILETYPE_FILE  [extern]
```

4.14.4.14 FILETYPE_LINK

```
const int FILETYPE_LINK  [extern]
```

4.14.4.15 FILETYPE_UNKNOWN

```
const int FILETYPE_UNKNOWN  [extern]
```

4.14.4.16 LASTBLOCK

```
const int LASTBLOCK  [extern]
```

4.14.4.17 ROOTDIR

```
const int ROOTDIR  [extern]
```

4.15 src/pennfat/pennfat.c File Reference

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <time.h>
#include <unistd.h>
#include <limits.h>
#include <errno.h>
#include "fat.h"
#include "safe.h"
#include "../util/parser.h"
#include "../util/util.h"
```

Include dependency graph for pennfat.c:

Macros

- #define CONTINUE {free(command); continue;}

Functions

- bool correct_argc (int expected, int actual)
- bool valid_fs_mounted (int fs_fd)
- bool all_files_exist (uint16_t *fat, int fs_fd, char *argv[ ], int first_file_arg, int last_file_arg)
- int main (int argc, char *argv[ ])

4.15.1 Macro Definition Documentation

4.15.1.1 CONTINUE

```
#define CONTINUE {free(command); continue;}
```

4.15.2 Function Documentation

4.15.2.1 all_files_exist()

```
bool all_files_exist (
            uint16_t * fat,
            int fs_fd,
            char * argv[],
            int first_file_arg,
            int last_file_arg )
```

4.15.2.2 correct_argc()

```
bool correct_argc (
            int expected,
            int actual )
```

4.15.2.3 main()

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

4.15.2.4 valid_fs_mounted()

```
bool valid_fs_mounted (
            int fs_fd )
```

4.16 src/pennfat/safe.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/types.h>
#include "safe.h"
```

Include dependency graph for safe.c:

Functions

- int safe_open (const char *pathname, int flags, mode_t mode)
- void safe_close (int fd)
- int safe_read (int fd, void *buf, size_t count)
- void safe_write (int fd, const void *buf, size_t count)
- off_t safe_lseek (int fd, off_t offset, int whence)
- void safe_msync (void *addr, size_t length, int flags)
- void * safe_mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset)
- void safe_munmap (void *addr, size_t length)

4.16.1 Function Documentation

4.16.1.1 safe_close()

```
void safe_close (
            int fd )
```

Closes a file descriptor with error handling.

Parameters

fd	The file descriptor to close.

Returns

None.

4.16.1.2 safe_lseek()

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

Performs a seek operation on a file descriptor with error handling.

Parameters

fd	The file descriptor to seek.
offset	The offset to move the file pointer.
whence	The reference point for the offset.

Returns

Returns the resulting offset location if successful; otherwise, exits with an error message.

4.16.1.3 safe_mmap()

```
void* safe_mmap (
            void * addr,
            size_t length,
            int prot,
            int flags,
            int fd,
            off_t offset )
```

Maps files or devices into memory with error handling.

Parameters

addr	The desired starting address for the mapping.
length	The length of the mapping.
prot	The desired memory protection of the mapping.
flags	The type of the mapping.
fd	The file descriptor of the file to map.
offset	The offset within the file to start the mapping.

Returns

The starting address of the mapped region.

4.16.1.4 safe_msync()

```
void safe_msync (
            void * addr,
            size_t length,
            int flags )
```

Synchronizes changes to a file mapping with error handling.

Parameters

addr	The starting address of the file mapping.
length	The length of the file mapping.
flags	Flags specifying the type of synchronization.

Returns

None.

4.16.1.5 safe_munmap()

```
void safe_munmap (
            void * addr,
            size_t length )
```

Unmaps files or devices from memory with error handling.

Parameters

addr	The starting address of the mapped region.
length	The length of the mapped region.

4.16.1.6 safe_open()

```
int safe_open (
            const char * pathname,
            int flags,
            mode_t mode )
```

Opens a file with error handling.

Parameters

pathname	The path of the file to open.
flags	Flags specifying the mode of opening.
mode	The permissions to set if the file is created.

Returns

Returns the file descriptor if successful; otherwise, exits with an error message.

4.16.1.7 safe_read()

```
int safe_read (
            int fd,
            void * buf,
            size_t count )
```

Reads data from a file descriptor with error handling.

Parameters

fd	The file descriptor to read from.
buf	The buffer to store the read data.
count	The number of bytes to read.

Returns

Returns the number of bytes read if successful; otherwise, exits with an error message.

4.16.1.8 safe_write()

```
void safe_write (
            int fd,
            const void * buf,
            size_t count )
```

Writes data to a file descriptor with error handling.

Parameters

fd	The file descriptor to write to.
buf	The buffer containing the data to write.
count	The number of bytes to write.

Returns

None.

4.17 src/pennfat/safe.h File Reference

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

Functions

- int safe_open (const char *, int, mode_t)
- void safe_close (int)
- int safe_read (int, void *, size_t)
- void safe_write (int, const void *, size_t)
- off_t safe_lseek (int fd, off_t offset, int whence)
- void safe_msync (void *addr, size_t length, int flags)
- void * safe_mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset)
- void safe_munmap (void *addr, size_t length)

4.17.1 Function Documentation

4.17.1.1 safe_close()

```
void safe_close (
            int fd )
```

Closes a file descriptor with error handling.

Parameters

fd	The file descriptor to close.

Returns

None.

4.17.1.2 safe_lseek()

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

Performs a seek operation on a file descriptor with error handling.

Parameters

fd	The file descriptor to seek.
offset	The offset to move the file pointer.
whence	The reference point for the offset.

Returns

Returns the resulting offset location if successful; otherwise, exits with an error message.

4.17.1.3 safe_mmap()

```
void* safe_mmap (
            void * addr,
            size_t length,
```

```
            int prot,
            int flags,
            int fd,
            off_t offset )
```

Maps files or devices into memory with error handling.

Parameters

addr	The desired starting address for the mapping.
length	The length of the mapping.
prot	The desired memory protection of the mapping.
flags	The type of the mapping.
fd	The file descriptor of the file to map.
offset	The offset within the file to start the mapping.

Returns

The starting address of the mapped region.

4.17.1.4 safe_msync()

```
void safe_msync (
            void * addr,
            size_t length,
            int flags )
```

Synchronizes changes to a file mapping with error handling.

Parameters

addr	The starting address of the file mapping.
length	The length of the file mapping.
flags	Flags specifying the type of synchronization.

Returns

None.

4.17.1.5 safe_munmap()

```
void safe_munmap (
            void * addr,
            size_t length )
```

Unmaps files or devices from memory with error handling.

Parameters

addr	The starting address of the mapped region.
length	The length of the mapped region.

4.17.1.6 safe_open()

```
int safe_open (
            const char * pathname,
            int flags,
            mode_t mode )
```

Opens a file with error handling.

Parameters

pathname	The path of the file to open.
flags	Flags specifying the mode of opening.
mode	The permissions to set if the file is created.

Returns

Returns the file descriptor if successful; otherwise, exits with an error message.

4.17.1.7 safe_read()

```
int safe_read (
            int fd,
            void * buf,
            size_t count )
```

Reads data from a file descriptor with error handling.

Parameters

fd	The file descriptor to read from.
buf	The buffer to store the read data.
count	The number of bytes to read.

Returns

Returns the number of bytes read if successful; otherwise, exits with an error message.

4.17.1.8 safe_write()

```
void safe_write (
            int fd,
            const void * buf,
            size_t count )
```

Writes data to a file descriptor with error handling.

Parameters

fd	The file descriptor to write to.
buf	The buffer containing the data to write.
count	The number of bytes to write.

Returns

None.

4.18 src/pennos.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include "util/globals.h"
#include "shell/pennos-shell.h"
#include "pennfat/fat.h"
#include "kernel/PCB.h"
#include "kernel/scheduler.h"
#include "logger/logger.h"
```

Include dependency graph for pennos.c:

Functions

- int main (int argc, char ∗argv[ ])

4.18.1 Function Documentation

4.18.1.1 main()

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

Entry point for PennOS. Initializes the logger, filesystem, and spawns the shell process.

Parameters

argc	The number of command-line arguments.
argv	An array of command-line arguments.

Returns

Returns 1 if the number of command-line arguments is less than 2.

4.19 src/shell/job-list.c File Reference

```
#include <stdlib.h>
#include <stdio.h>
#include "job-list.h"
#include "../util/util.h"
#include "../util/p-errno.h"
#include "../filesystem/filesystem.h"
#include "../kernel/puser-functions.h"
```

Include dependency graph for job-list.c:

Functions

- job_t * job_find_by_jobid (job_t **head, int target_job_id)
- int job_get_last (job_t **head)
- job_t * jobs_push (job_t **head, int job_id, int pid, int stop_order)
- void jobs_insert (job_t **head, job_t *new_job)
- job_t * jobs_remove (job_t **head, int target_job_id)
- void job_print (job_t *job)

4.19.1 Function Documentation

4.19.1.1 job_find_by_jobid()

```
job_t* job_find_by_jobid (
            job_t ** head,
            int target_job_id )
```

Finds a job by its job ID in the linked list of jobs.

Parameters

head	A pointer to the head of the linked list of jobs.
target_job↩ _id	The job ID to search for.

Returns

Returns a pointer to the found job or NULL if not found.

4.19.1.2 job_get_last()

```
int job_get_last (
            job_t ** head )
```

Gets the job ID of the last job in the linked list.

Parameters

head	A pointer to the head of the linked list of jobs.

Returns

Returns the job ID of the last job or -1 if the list is empty.

4.19.1.3 job_print()

```
void job_print (
            job_t * job )
```

Prints information about a job.

Parameters

job	A pointer to the job structure to be printed.

4.19.1.4 jobs_insert()

```
void jobs_insert (
            job_t ** head,
            job_t * new_job )
```

Inserts a new job into the linked list in ascending order based on job ID.

Parameters

head	A pointer to the head of the linked list of jobs.
new_job	A pointer to the new job to be inserted.

4.19.1.5 jobs_push()

```
job_t* jobs_push (
            job_t ** head,
            int job_id,
            int pid,
            int stop_order )
```

Pushes a new job to the end of the linked list.

Parameters

head	A pointer to the head of the linked list of jobs.
job_id	The job ID of the new job.
pid	The process ID associated with the job.
stop_order	The order in which the job should be stopped.

Returns

Returns a pointer to the newly added job.

4.19.1.6 jobs_remove()

```
job_t* jobs_remove (
            job_t ** head,
            int target_job_id )
```

Removes a job with the specified job ID from the linked list.

Parameters

head	A pointer to the head of the linked list of jobs.
target_job↩ _id	The job ID of the job to be removed.

Returns

A pointer to the removed job, or NULL if not found.

4.20 src/shell/job-list.h File Reference

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

Include dependency graph for job-list.h: This graph shows which files directly or indirectly include this file:

Data Structures

- struct job

Macros

- #define NOT_STOPPED -1

Typedefs

- typedef struct job job_t

Functions

- job_t * job_find_by_jobid (job_t **head, int target_job_id)
- int job_get_last (job_t **head)
- job_t * jobs_push (job_t **head, int job_id, int pid, int stop_order)
- void jobs_insert (job_t **head, job_t *new_job)
- job_t * jobs_remove (job_t **head, int target_job_id)
- void job_print (job_t *job)

4.20.1 Macro Definition Documentation

4.20.1.1 NOT_STOPPED

```
#define NOT_STOPPED -1
```

4.20.2 Typedef Documentation

4.20.2.1 job_t

```
typedef struct job job_t
```

4.20.3 Function Documentation

4.20.3.1 job_find_by_jobid()

```
job_t* job_find_by_jobid (
            job_t ** head,
            int target_job_id )
```

find a job struct by job_id

Parameters

head	the job linkedlist
target_job↩ _id	the job_id to find

Returns

the job struct if it was found, or `NULL` otherwise

Finds a job by its job ID in the linked list of jobs.

Parameters

head	A pointer to the head of the linked list of jobs.
target_job↩ _id	The job ID to search for.

Returns

Returns a pointer to the found job or NULL if not found.

4.20.3.2 job_get_last()

```
int job_get_last (
            job_t ** head )
```

get last process (most recent background process)

Parameters

head	the job linkedlist

Returns

the last job_id, or `-1` if the list is empty

Gets the job ID of the last job in the linked list.

Parameters

head	A pointer to the head of the linked list of jobs.

Returns

Returns the job ID of the last job or -1 if the list is empty.

4.20.3.3 job_print()

```
void job_print (
            job_t * job )
```

DEBUG: print job info

Parameters

job	the job

Prints information about a job.

Parameters

job	A pointer to the job structure to be printed.

4.20.3.4 jobs_insert()

```
void jobs_insert (
            job_t ** head,
            job_t * new_job )
```

insert a job struct at the correct position to maintain sortedness by job_id

Parameters

head	the job linkedlist
new_job	the job struct

Returns

none

Inserts a new job into the linked list in ascending order based on job ID.

Parameters

head	A pointer to the head of the linked list of jobs.
new_job	A pointer to the new job to be inserted.

4.20.3.5 jobs_push()

```
job_t* jobs_push (
            job_t ** head,
            int job_id,
            int pid,
            int stop_order )
```

create a job and push it onto the linkedlist

Parameters

head	the job linkedlist
job_id	the new job_id
pid	the job's pid mapping
stop_order	`-1` or `NOT_STOPPED` if foreground process; main `stop_order` if background process

Returns

the created job struct

Pushes a new job to the end of the linked list.

Parameters

head	A pointer to the head of the linked list of jobs.
job_id	The job ID of the new job.
pid	The process ID associated with the job.
stop_order	The order in which the job should be stopped.

Returns

Returns a pointer to the newly added job.

4.20.3.6 jobs_remove()

```
job_t* jobs_remove (
            job_t ** head,
            int target_job_id )
```

remove the specified job and return it

Parameters

head	the job linkedlist
target_job↩ _id	the job_id to find

Returns

the job struct if it was found, or `NULL` otherwise

Removes a job with the specified job ID from the linked list.

Parameters

head	A pointer to the head of the linked list of jobs.
target_job↩ _id	The job ID of the job to be removed.

Returns

A pointer to the removed job, or NULL if not found.

4.21 src/shell/pennos-shell.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include "../util/parser.h"
#include "../util/util.h"
#include "../util/globals.h"
#include "../util/p-errno.h"
#include "../util/safe-user.h"
#include "../filesystem/filesystem.h"
#include "../kernel/puser-functions.h"
#include "../logger/logger.h"
#include "job-list.h"
```

Include dependency graph for pennos-shell.c:

Macros

- #define PROMPT "$ "
- #define CONTINUE {free(command); continue;}

Functions

- void stop_handler (int signal)
- void term_handler (int signal)
- void shell_cat (int argc, char ∗argv[ ])
- void shell_sleep (int argc, char ∗argv[ ])
- void shell_busy (int argc, char ∗argv[ ])
- void shell_echo (int argc, char ∗argv[ ])
- void shell_ls (int argc, char ∗argv[ ])
- void shell_touch (int argc, char ∗argv[ ])
- void shell_mv (int argc, char ∗argv[ ])
- void shell_cp (int argc, char ∗argv[ ])
- void shell_rm (int argc, char ∗argv[ ])
- void shell_chmod (int argc, char ∗argv[ ])
- void shell_ps (int argc, char ∗argv[ ])

- void shell_kill (int argc, char ∗argv[ ])
- void zombie_child ()
- void shell_zombify (int argc, char ∗argv[ ])
- void orphan_child ()
- void shell_orphanify (int argc, char ∗argv[ ])
- void cull_helper (job_t ∗job)
- void cull_background ()
- void debug_print_jobs ()
- void empty_reaped ()
- int spawn_command (char ∗command[ ], int in_fd, int out_fd)
- int execute_command (char ∗command[ ], const char ∗in_filename, const char ∗out_filename, bool append↩ _mode)
- int execute_script (char ∗command_in[ ], const char ∗in_filename, const char ∗out_filename, bool append↩ _mode)
- void pennos_shell (int argc, char ∗argv[ ])

Variables

- const int MAX_ARGUMENTS = 1000
- const char ∗ MAN_COMMANDS
- job_t ∗ foreground_job = NULL
- int current_jobid
- int jobid_ctr = 1
- int stop_order = 1
- job_t ∗ head = NULL
- job_t ∗∗ background = &head
- int n_reaped = 0
- job_t ∗ reaped [1000]
- bool stop_trigger = false

4.21.1 Macro Definition Documentation

4.21.1.1 CONTINUE

```
#define CONTINUE {free(command); continue;}
```

4.21.1.2 PROMPT

```
#define PROMPT "$ "
```

4.21.2 Function Documentation

4.21.2.1 cull_background()

```
void cull_background ( )
```

4.21.2.2 cull_helper()

```
void cull_helper (
            job_t * job )
```

4.21.2.3 debug_print_jobs()

```
void debug_print_jobs ( )
```

4.21.2.4 empty_reaped()

```
void empty_reaped ( )
```

4.21.2.5 execute_command()

```
int execute_command (
            char * command[],
            const char * in_filename,
            const char * out_filename,
            bool append_mode )
```

start a (single) independent pennOS process & return the pid; does not wait for the process

Parameters

command	the command to parse & execute
in_filename	`NULL` for `F_STDIN`, otherwise the input file
out_filename	`NULL` for `F_STDOUT`, otherwise the output file
append_mode	whether to append to `out_filename`; ignore if `out_filename == NULL`

Returns

the pid of the spawned process

4.21.2.6 execute_script()

```
int execute_script (
            char * command_in[],
            const char * in_filename,
            const char * out_filename,
            bool append_mode )
```

4.21.2.7 orphan_child()

```
void orphan_child ( )
```

4.21.2.8 pennos_shell()

```
void pennos_shell (
            int argc,
            char * argv[] )
```

shell function to be called upon startup

Parameters

argc	TODO: does this need any args?
argv	TODO

Returns

nonzero on error, 1 on success

4.21.2.9 shell_busy()

```
void shell_busy (
            int argc,
            char * argv[] )
```

4.21.2.10 shell_cat()

```
void shell_cat (
            int argc,
            char * argv[] )
```

4.21.2.11 shell_chmod()

```
void shell_chmod (
            int argc,
            char * argv[] )
```

4.21.2.12 shell_cp()

```
void shell_cp (
            int argc,
            char * argv[] )
```

4.21.2.13 shell_echo()

```
void shell_echo (
            int argc,
            char * argv[] )
```

4.21.2.14 shell_kill()

```
void shell_kill (
            int argc,
            char * argv[] )
```

4.21.2.15 shell_ls()

```
void shell_ls (
            int argc,
            char * argv[] )
```

4.21.2.16 shell_mv()

```
void shell_mv (
            int argc,
            char * argv[] )
```

4.21.2.17 shell_orphanify()

```
void shell_orphanify (
            int argc,
            char * argv[] )
```

4.21.2.18 shell_ps()

```
void shell_ps (
            int argc,
            char * argv[] )
```

4.21.2.19 shell_rm()

```
void shell_rm (
            int argc,
            char * argv[] )
```

4.21.2.20 shell_sleep()

```
void shell_sleep (
            int argc,
            char * argv[] )
```

4.21.2.21 shell_touch()

```
void shell_touch (
            int argc,
            char * argv[] )
```

4.21.2.22 shell_zombify()

```
void shell_zombify (
            int argc,
            char * argv[] )
```

4.21.2.23 spawn_command()

```
int spawn_command (
            char * command[],
            int in_fd,
            int out_fd )
```

4.21.2.24 stop_handler()

```
void stop_handler (
            int signal )
```

4.21.2.25 term_handler()

```
void term_handler (
            int signal )
```

4.21.2.26 zombie_child()

```
void zombie_child ( )
```

4.21.3 Variable Documentation

4.21.3.1 background

```
job_t** background = &head
```

4.21.3.2 current_jobid

```
int current_jobid
```

4.21.3.3 foreground_job

```
job_t* foreground_job = NULL
```

4.21.3.4 head

```
job_t* head = NULL
```

4.21.3.5 jobid_ctr

```
int jobid_ctr = 1
```

4.21.3.6 MAN_COMMANDS

```
const char* MAN_COMMANDS
```

Initial value:

```
= "\
--- Independently scheduled processes ---\n\
cat OUTPUT\n\
sleep SECONDS\n\
busy\n\
echo [ STRING ]\n\
ls [ FILENAME ]\n\
touch FILE ...\n\
mv SRC DEST\n\
cp SRC DEST\n\
rm FILE ...\n\
chmod FILE PERM\n\
ps\n\
kill [ -SIGNAL_NAME ] PID ...\n\
zombify\n\
orphanify\n\
\n\
--- Shell subroutines ---\n\
nice PRIORITY COMMAND [ ARG ]\n\
nice_pid PRIORITY PID\n\
man\n\
bg [ JOB_ID ]\n\
fg [ JOB_ID ]\n\
jobs\n\
logout\n\
"
```

4.21.3.7 MAX_ARGUMENTS

```
const int MAX_ARGUMENTS = 1000
```

4.21.3.8 n_reaped

```
int n_reaped = 0
```

4.21.3.9 reaped

```
job_t* reaped[1000]
```

4.21.3.10 stop_order

```
int stop_order = 1
```

4.21.3.11 stop_trigger

```
bool stop_trigger = false
```

4.22 src/shell/pennos-shell.h File Reference

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

Functions

- void pennos_shell (int argc, char *argv[ ])

4.22.1 Function Documentation

4.22.1.1 pennos_shell()

```
void pennos_shell (
            int argc,
            char * argv[] )
```

shell function to be called upon startup

Parameters

argc	TODO: does this need any args?
argv	TODO

Returns

nonzero on error, 1 on success

4.23 src/util/globals.c File Reference

```
#include <stdint.h>
```
Include dependency graph for globals.c:

Variables

- int fs_fd
- uint16_t * fat

4.23.1 Variable Documentation

4.23.1.1 fat

```
uint16_t* fat
```

4.23.1.2 fs_fd

```
int fs_fd
```

4.24 src/util/globals.h File Reference

```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
```
Include dependency graph for globals.h: This graph shows which files directly or indirectly include this file:

Macros

- #define S_SIGSTOP 123 /* a thread receiving this signal should be stopped */
- #define S_SIGCONT 456 /* a thread receiving this signal should be continued */
- #define S_SIGTERM 789 /* a thread receiving this signal should be terminated */
- #define S_SIGCHLD 812 /* a thread receiving this signal should be terminated */
- #define T_RUNNING 111
- #define T_STOPPED 222
- #define T_BLOCKED 333
- #define T_ZOMBIED 444

Variables

- int fs_fd
- uint16_t * fat

4.24.1 Macro Definition Documentation

4.24.1.1 S_SIGCHLD

```
#define S_SIGCHLD 812 /* a thread receiving this signal should be terminated */
```

4.24.1.2 S_SIGCONT

```
#define S_SIGCONT 456 /* a thread receiving this signal should be continued */
```

4.24.1.3 S_SIGSTOP

```
#define S_SIGSTOP 123 /* a thread receiving this signal should be stopped */
```

4.24.1.4 S_SIGTERM

```
#define S_SIGTERM 789 /* a thread receiving this signal should be terminated */
```

4.24.1.5 T_BLOCKED

```
#define T_BLOCKED 333
```

4.24.1.6 T_RUNNING

```
#define T_RUNNING 111
```

4.24.1.7 T_STOPPED

```
#define T_STOPPED 222
```

4.24.1.8 T_ZOMBIED

```
#define T_ZOMBIED 444
```

4.24.2 Variable Documentation

4.24.2.1 fat

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

4.24.2.2 fs_fd

```
int fs_fd  [extern]
```

4.25 src/util/p-errno.c File Reference

```
#include <stdio.h>
#include "p-errno.h"
#include "util.h"
#include "../filesystem/filesystem.h"
```

Include dependency graph for p-errno.c:

Functions

- char * err_string (int errno)
- void p_perror (const char *message)

Variables

- int ERRNO = ERR_NONE

4.25.1 Function Documentation

4.25.1.1 err_string()

```
char* err_string (
            int errno )
```

Returns a string description for the given error code.

Parameters

errno	The error code.

Returns

A string description for the error code.

4.25.1.2 p_perror()

```
void p_perror (
            const char * message )
```

Print an error message along with the corresponding error string.

Parameters

message	The additional message to print.

4.25.2 Variable Documentation

4.25.2.1 ERRNO

```
int ERRNO = ERR_NONE
```

4.26 src/util/p-errno.h File Reference

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

Macros

- #define ERR_NONE 0
- #define ERR_FS_FILE_NOT_FOUND 1000
- #define ERR_F_OPEN_INVALID_PERMS 1010
- #define ERR_F_OPEN_WRITE_INUSE 1011
- #define ERR_F_OPEN_CREATE_READ 1012
- #define ERR_F_OPEN_INVALID_MODE 1013
- #define ERR_F_READ_TERM_OUT 1020
- #define ERR_F_WRITE_TERM_IN 1030
- #define ERR_F_WRITE_RONLY 1031
- #define ERR_F_CLOSE_TERMINAL 1040
- #define ERR_F_UNLINK_NOT_FOUND 1050
- #define ERR_F_LSEEK_TERMINAL 1060
- #define ERR_F_LSEEK_OOB 1061
- #define ERR_P_SPAWN_NULL_CHILD 2000
- #define ERR_P_SPAWN_NULL_STACK 2001
- #define ERR_P_WAITPID_NULL_CHILD 2010
- #define ERR_P_KILL_NULL_PROCESS 2020
- #define ERR_P_NICE_NULL_PROCESS 2030

Functions

- void p_perror (const char *message)

4.26.1 Macro Definition Documentation

4.26.1.1 ERR_F_CLOSE_TERMINAL

```
#define ERR_F_CLOSE_TERMINAL 1040
```

4.26.1.2 ERR_F_LSEEK_OOB

```
#define ERR_F_LSEEK_OOB 1061
```

4.26.1.3 ERR_F_LSEEK_TERMINAL

```
#define ERR_F_LSEEK_TERMINAL 1060
```

4.26.1.4 ERR_F_OPEN_CREATE_READ

```
#define ERR_F_OPEN_CREATE_READ 1012
```

4.26.1.5 ERR_F_OPEN_INVALID_MODE

```
#define ERR_F_OPEN_INVALID_MODE 1013
```

4.26.1.6 ERR_F_OPEN_INVALID_PERMS

```
#define ERR_F_OPEN_INVALID_PERMS 1010
```

4.26.1.7 ERR_F_OPEN_WRITE_INUSE

```
#define ERR_F_OPEN_WRITE_INUSE 1011
```

4.26.1.8 ERR_F_READ_TERM_OUT

```
#define ERR_F_READ_TERM_OUT 1020
```

4.26.1.9 ERR_F_UNLINK_NOT_FOUND

```
#define ERR_F_UNLINK_NOT_FOUND 1050
```

4.26.1.10 ERR_F_WRITE_RONLY

```
#define ERR_F_WRITE_RONLY 1031
```

4.26.1.11 ERR_F_WRITE_TERM_IN

```
#define ERR_F_WRITE_TERM_IN 1030
```

4.26.1.12 ERR_FS_FILE_NOT_FOUND

```
#define ERR_FS_FILE_NOT_FOUND 1000
```

4.26.1.13 ERR_NONE

```
#define ERR_NONE 0
```

4.26.1.14 ERR_P_KILL_NULL_PROCESS

```
#define ERR_P_KILL_NULL_PROCESS 2020
```

4.26.1.15 ERR_P_NICE_NULL_PROCESS

```
#define ERR_P_NICE_NULL_PROCESS 2030
```

4.26.1.16 ERR_P_SPAWN_NULL_CHILD

```
#define ERR_P_SPAWN_NULL_CHILD 2000
```

4.26.1.17 ERR_P_SPAWN_NULL_STACK

```
#define ERR_P_SPAWN_NULL_STACK 2001
```

4.26.1.18 ERR_P_WAITPID_NULL_CHILD

```
#define ERR_P_WAITPID_NULL_CHILD 2010
```

4.26.2 Function Documentation

4.26.2.1 p_perror()

```
void p_perror (
            const char * message )
```

print a message describing the meaning of the value of ERRNO

Parameters

message	the message to print

Returns

none

Print an error message along with the corresponding error string.

Parameters

message	The additional message to print.

4.27 src/util/parser.h File Reference

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

Include dependency graph for parser.h: This graph shows which files directly or indirectly include this file:

Data Structures

- struct parsed_command

Macros

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

Functions

- int parse_command (const char ∗cmd_line, struct parsed_command ∗∗result)
- void print_parsed_command (const struct parsed_command ∗cmd)

4.27.1 Macro Definition Documentation

4.27.1.1 EXPECT_COMMANDS

```
#define EXPECT_COMMANDS 7
```

4.27.1.2 EXPECT_INPUT_FILENAME

```
#define EXPECT_INPUT_FILENAME 5
```

4.27.1.3 EXPECT_OUTPUT_FILENAME

```
#define EXPECT_OUTPUT_FILENAME 6
```

4.27.1.4 UNEXPECTED_AMPERSAND

```
#define UNEXPECTED_AMPERSAND 4
```

4.27.1.5 UNEXPECTED_FILE_INPUT

```
#define UNEXPECTED_FILE_INPUT 1
```

4.27.1.6 UNEXPECTED_FILE_OUTPUT

```
#define UNEXPECTED_FILE_OUTPUT 2
```

4.27.1.7 UNEXPECTED_PIPELINE

```
#define UNEXPECTED_PIPELINE 3
```

4.27.2 Function Documentation

4.27.2.1 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.

4.27.2.2 print_parsed_command()

```
void print_parsed_command (
            const struct parsed_command * cmd )
```

4.28 src/util/safe-user.c File Reference

```
#include <stdint.h>
#include "safe-user.h"
#include "p-errno.h"
#include "../kernel/puser-functions.h"
#include "../filesystem/filesystem.h"
```

Include dependency graph for safe-user.c:

Functions

- int safe_f_open (const char *fname, int mode)
- int safe_f_read (int fd, int n, char *buf)
- int safe_f_write (int fd, const char *str, int n)
- int safe_f_close (int fd)
- int safe_f_unlink (const char *fname)
- int safe_f_lseek (int fd, int offset, int whence)
- int safe_f_print (const char *str)

4.28.1 Function Documentation

4.28.1.1 safe_f_close()

```
int safe_f_close (
            int fd )
```

4.28.1.2 safe_f_lseek()

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

4.28.1.3 safe_f_open()

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

4.28.1.4 safe_f_print()

```
int safe_f_print (
            const char * str )
```

4.28.1.5 safe_f_read()

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

4.28.1.6 safe_f_unlink()

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

4.28.1.7 safe_f_write()

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

4.29 src/util/safe-user.h File Reference

`#include <stdint.h>`
Include dependency graph for safe-user.h: This graph shows which files directly or indirectly include this file:

Functions

- int safe_f_open (const char ∗fname, int mode)
- int safe_f_read (int fd, int n, char ∗buf)
- int safe_f_write (int fd, const char ∗str, int n)
- int safe_f_close (int fd)
- int safe_f_unlink (const char ∗fname)
- int safe_f_lseek (int fd, int offset, int whence)
- int safe_f_print (const char ∗str)

4.29.1 Function Documentation

4.29.1.1 safe_f_close()

```
int safe_f_close (
            int fd )
```

4.29.1.2 safe_f_lseek()

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

4.29.1.3 safe_f_open()

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

4.29.1.4 safe_f_print()

```
int safe_f_print (
            const char * str )
```

4.29.1.5 safe_f_read()

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

4.29.1.6 safe_f_unlink()

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

4.29.1.7 safe_f_write()

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

4.30 src/util/util.c File Reference

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

Include dependency graph for util.c:

Functions

- int get_argc (char ∗argv[ ])
- void ∗ safe_malloc (size_t size)
- void safe_signal (int signum, void(∗handler)(int))

Variables

- const int IOBUFFER_SIZE = 10000
- const int ERRBUFFER_SIZE = 1000

4.30.1 Function Documentation

4.30.1.1 get_argc()

```
int get_argc (
            char * argv[] )
```

Get the number of arguments in an array of strings.

Parameters

argv	The array of strings.

Returns

The number of arguments.

4.30.1.2 safe_malloc()

```
void* safe_malloc (
            size_t size )
```

Allocate memory using malloc.

Parameters

size	The size of the memory to allocate.

Returns

A pointer to the allocated memory.

4.30.1.3 safe_signal()

```
void safe_signal (
            int signum,
            void(*)(int) handler )
```

Set a signal handler using signal.

Parameters

signum	The signal number.
handler	The signal handler function.

4.30.2 Variable Documentation

4.30.2.1 ERRBUFFER_SIZE

```
const int ERRBUFFER_SIZE = 1000
```

4.30.2.2 IOBUFFER_SIZE

```
const int IOBUFFER_SIZE = 10000
```

4.31 src/util/util.h File Reference

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

Macros

- #define PRINT(x) {fprintf(stderr, "%d\n", x);}
- #define PRINTE {fprintf(stderr, "e\n");}

Functions

- int get_argc (char ∗argv[ ])
- void ∗ safe_malloc (size_t size)
- void safe_signal (int signum, void(∗handler)(int))

Variables

- const int IOBUFFER_SIZE
- const int ERRBUFFER_SIZE

4.31.1 Macro Definition Documentation

4.31.1.1 PRINT

```
#define PRINT(
              x ) {fprintf(stderr, "%d\n", x);}
```

4.31.1.2 PRINTE

```
#define PRINTE {fprintf(stderr, "e\n");}
```

4.31.2 Function Documentation

4.31.2.1 get_argc()

```
int get_argc (
             char * argv[] )
```

Get the number of arguments in an array of strings.

Parameters

argv	The array of strings.

Returns

The number of arguments.

4.31.2.2 safe_malloc()

```
void* safe_malloc (
             size_t size )
```

Allocate memory using malloc.

Parameters

size	The size of the memory to allocate.

Returns

A pointer to the allocated memory.

4.31.2.3 safe_signal()

```
void safe_signal (
            int signum,
            void(*)(int) handler )
```

Set a signal handler using signal.

Parameters

signum	The signal number.
handler	The signal handler function.

4.31.3 Variable Documentation

4.31.3.1 ERRBUFFER_SIZE

```
const int ERRBUFFER_SIZE  [extern]
```

4.31.3.2 IOBUFFER_SIZE

```
const int IOBUFFER_SIZE  [extern]
```

Index