PennOS

Generated by Doxygen 1.9.1

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

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

Chapter 3

Class Documentation

3.1 entry_location Struct Reference

Logical location of a file entry in a directory block.

```
#include <fat_tables.h>
```

Public Attributes

- uint16_t block_num

  Block number containing the entry.
- int entry_index

  Index within the block (0-based)
- bool exists

  Indicates if the entry already exists.

3.1.1 Detailed Description

Logical location of a file entry in a directory block.

Used for locating or updating a specific file entry on disk.

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

- src/pennfat/fat_tables.h

3.2 file_descriptor_t Struct Reference

In-memory metadata for open files (used by kernel)

```
#include <fat_tables.h>
```

Collaboration diagram for file_descriptor_t:

Public Attributes

- bool in_use

  Whether this descriptor is in use.
- char filename [32]

  Name of the open file.
- int mode

  Open mode (MODE_READ / MODE_WRITE / MODE_APPEND)
- uint32_t offset

  Current byte offset within the file.
- uint32_t size

  Cached file size.
- uint16_t first_block

  First block of the file.
- uint16_t current_block

  Current data block being accessed.
- uint16_t block_offset

  Offset within the current block.
- entry_location_t entry_location

  Location of the file's directory entry.

3.2.1 Detailed Description

In-memory metadata for open files (used by kernel)

Represents the runtime state of a file descriptor opened by a process or shell.

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

- src/pennfat/fat_tables.h

3.3 file_entry Struct Reference

On-disk representation of a file entry (64 bytes total)

```
#include <fat_tables.h>
```

Public Attributes

- char name [32]

  File name (null-terminated if shorter)
- uint32_t size

  File size in bytes.
- uint16_t firstBlock

  Index of first data block.
- uint8_t type

  FILE_TYPE_REGULAR or FILE_TYPE_DIRECTORY.
- uint8_t perm

  Permissions using PERM_∗ flags.
- time_t mtime

  Last modified timestamp.
- char reserved [16]

  Reserved for future use or padding.

3.3.1 Detailed Description

On-disk representation of a file entry (64 bytes total)

Each file entry represents metadata for one file in the directory. All entries are stored packed into directory blocks.

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

- src/pennfat/fat_tables.h

3.4 file_system Struct Reference

In-memory structure for mounted file system metadata.

```
#include <fat_tables.h>
```

Public Attributes

- uint16_t * fat_table

    Pointer to loaded FAT block array.
- int fs_fd

    File descriptor to disk image.
- uint32_t fat_size

    Number of FAT entries.
- uint32_t num_blocks

    Total number of blocks in disk image.
- uint32_t block_size

    Size of each block in bytes.
- bool is_mounted

    Whether file system is currently active.
- char fs_name [32]

    Mounted file system name.
- uint32_t fat_offset

    Byte offset to FAT table.
- uint32_t data_offset

    Byte offset to first data block.

3.4.1 Detailed Description

In-memory structure for mounted file system metadata.

Represents the active PennFAT file system. Maintained in global memory after mounting.

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

- src/pennfat/fat_tables.h

3.5 job_st Struct Reference

Public Attributes

- pid_t pgid

  Process group ID of the job.
- int job_id

  Internal job ID used by the shell.
- char * cmd

  Full command string.
- char status

  Job status: 'r' (running), 's' (stopped)

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

- src/shell/jobs.h

3.6 job_t Struct Reference

Represents a shell job tracked via PGID.

```
#include <jobs.h>
```

3.6.1 Detailed Description

Represents a shell job tracked via PGID.

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

- src/shell/jobs.h

3.7 parsed_command Struct Reference

Represents a parsed command line with optional I/O redirection, backgrounding, and pipelines.

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

Public Attributes

- bool is_background

  True if command ends with '&' (background job)
- bool is_file_append

  True if output is redirected with '>>'.
- const char * stdin_file

  Filename for input redirection ('<'), or NULL.
- const char * stdout_file

  Filename for output redirection ('>' or '>>'), or NULL.
- size_t num_commands

  Number of pipeline stages.
- char ** commands [ ]

  An array of command argument vectors.

3.7.1 Detailed Description

Represents a parsed command line with optional I/O redirection, backgrounding, and pipelines.

This structure is dynamically allocated by `parse_command()` and contains all necessary information to execute the parsed shell command(s).

3.7.2 Member Data Documentation

3.7.2.1 commands

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

An array of command argument vectors.

`commands[i]` is a `char*[]` representing one pipeline stage, and each is null-terminated (i.e., like `argv[]`).

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

- src/shell/parser.h

3.8 pcb Struct Reference

The Process Control Block (PCB)

```
#include <pcb.h>
```

Collaboration diagram for pcb:

Public Attributes

- pid_t pid
    Process ID.
- pid_t pgid
    Process group ID.
- Priority priority
    Scheduling priority.
- pid_t parent_pid
    Parent process ID.
- Vec children_pids
    List of child PIDs.
- spthread_t ∗ thread
    Associated kernel thread.
- Status status
    Current status of process.
- Vec file_descriptors

    Open file descriptor table.
- char ∗ cmd_name

    Name of the command.
- pid_t waitpid

    PID this process is waiting on.
- Vec zombie_children

    List of zombie child PIDs.
- Vec signaled_children

    List of signaled child PIDs.
- int sleep_time

    Remaining sleep ticks; -1 if not sleeping.
- ExitStatus exit_status

    How the process exited or changed state.

3.8.1 Detailed Description

The Process Control Block (PCB)

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

- src/kernel/pcb.h

3.9 proc_fd Struct Reference

Represents a file descriptor opened by a process.

```
#include <fd.h>
```

Public Attributes

- int **fd**

3.9.1 Detailed Description

Represents a file descriptor opened by a process.

This structure wraps a file descriptor and can be passed between modules.

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

- src/kernel/fd.h

3.10 process_info Struct Reference

Public Attributes

- pid_t **pid**
- pid_t **parent_pid**
- int **priority**
- char **status**
- char ∗ **cmd**

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

- src/util/interface.h

3.11 process_info_t Struct Reference

Holds basic information about a process.

```
#include <interface.h>
```

3.11.1 Detailed Description

Holds basic information about a process.

Fields:

- pid: Process ID
- parent_pid: Parent process ID
- priority: Priority level of the process (0-2)
- status: One of {'R': running, 'S': sleeping, 'Z': zombie, 'T': stopped}
- cmd: Pointer to command string (dynamically allocated)

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

- src/util/interface.h

3.12 routine_args Struct Reference

Argument structure passed to all spthread-executed routines.

```
#include <routines.h>
```

3.12.1 Detailed Description

Argument structure passed to all spthread-executed routines.

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

- src/shell/routines.h

3.13 routine_args_st Struct Reference

Public Attributes

- bool is_background

  Whether the command should run in the background.
- int priority

  Scheduling priority (lower = higher priority)
- int input_fd

  Input file descriptor.
- int output_fd

  Output file descriptor.
- spthread_fn actual_routine

  Function pointer to the actual routine.
- char ∗∗ actual_arg

  Null-terminated array of string arguments.
- void ∗ **actual_arg**

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

- src/shell/routines.h
- src/sys_call/sys_call.c

3.14 spthread_fwd_args_st Struct Reference

Collaboration diagram for spthread_fwd_args_st:

Public Attributes

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

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

- src/util/spthread.c

3.15 spthread_meta_st Struct Reference

Public Attributes

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

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

- src/util/spthread.c

3.16 spthread_signal_args_st Struct Reference

Public Attributes

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

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

- src/util/spthread.c

3.17 spthread_st Struct Reference

Collaboration diagram for spthread_st:

Public Attributes

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

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

- src/util/spthread.h

3.18 vec_st Struct Reference

Public Attributes

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

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

- src/util/Vec.h

Chapter 4

File Documentation

4.1 src/kernel/error.h File Reference

Defines common system error codes and error handling interfaces.

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

4.2 src/kernel/kernel.h File Reference

Core interface for process and file system operations in the PennOS kernel.

```
#include "fd.h"
#include "pcb.h"
```

Include dependency graph for kernel.h:

Functions

- void k_init_kernel (int log_fd)

  Initialize the kernel and core data structures.
- void k_start_kernel ()

  Start the kernel scheduler.
- pcb_t ∗ k_proc_create (pcb_t ∗parent, char ∗name)

  Create a new child process, inheriting applicable properties from the parent.
- void k_proc_start (pid_t pid)

  Start a newly created process.
- void k_enter_kernel_mode ()

  Disable interrupts.
- void k_exit_kernel_mode ()

  Enable interrupts.
- pcb_t ∗ k_get_calling_pcb ()

  Get the PCB of the calling process.
- int k_nice (pid_t pid, Priority priority)

  Set the priority of the specified thread.
- int k_open (const char ∗fname, FileDescriptorMode mode)

open a file name fname with the mode mode and return a file descriptor.
- int k_write (int fd, const char ∗str, int n)

  write n bytes of the string referenced by str to the file fd and increment the file pointer by n. On return, k_write returns the number of bytes written, or -1 on error.
- int k_read (int fd, char ∗buf, int n)

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

  close the file fd and return 0 on success, or a negative value on failure.
- int k_ps (Vec ∗vec)

  List all processes, store them as process_info_t pointer to vec.
- void k_sleep (unsigned int ticks)

  Suspends execution of the calling proces for a specified number of clock ticks.
- int k_kill (pid_t pid, int sig_num)

  Send signal specified by sig_num to pid.
- pid_t k_waitpid (pid_t pid, int ∗wstatus, bool nohang)

  Wait on a child of the calling process, until it changes state. If `nohang` is true, this will not block the calling process and return immediately.
- void k_exit (pid_t pid)

  Unconditionally exit the calling process.
- void k_shutdown ()

  Stop the scheduler and release all resources used by kernel.
- void k_set_foreground_pgid (pid_t pgid)

  Set the foreground pgid, this is used for signaling.
- int k_setpgid (pid_t pid, pid_t pgid)

  Set the PGID of pid.

4.2.1 Detailed Description

Core interface for process and file system operations in the PennOS kernel.

This header defines system-level APIs for process management, file descriptor handling, signal delivery, scheduling control, and OS lifecycle management.

4.2.2 Function Documentation

4.2.2.1 k_close()

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

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

Parameters

fd	the file descriptor num

Returns

0 on success, -1 on error

4.2.2.2 k_enter_kernel_mode()

```
void k_enter_kernel_mode ( )
```

Disable interrupts.

Disable interrupts.

4.2.2.3 k_exit_kernel_mode()

```
void k_exit_kernel_mode ( )
```

Enable interrupts.

Enable interrupts.

4.2.2.4 k_get_calling_pcb()

```
pcb_t* k_get_calling_pcb ( )
```

Get the PCB of the calling process.

Returns

A pointer to the PCB of the calling process, NULL on error

4.2.2.5 k_init_kernel()

```
void k_init_kernel (
             int log_fd )
```

Initialize the kernel and core data structures.

This function sets up essential queues and initializes the scheduler and logger. It should be called once at OS startup.

Parameters

log↩_fd	File descriptor used for kernel event logging.

4.2.2.6 k_kill()

```
int k_kill (
            pid_t pid,
            int sig_num )
```

Send signal specified by sig_num to pid.

Parameters

pid	Positive number for a process, negative number for a process group

4.2.2.7 k_nice()

```
int k_nice (
            pid_t pid,
            Priority priority )
```

Set the priority of the specified thread.

Parameters

pid	Process ID of the target thread.
priority	The new priority value of the thread (0, 1, or 2)

Returns

0 on success, -1 on failure.

4.2.2.8 k_open()

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

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

Parameters

fname	name of the file
mode	F_WRITE - writing and reading, truncates if the file exists, or creates it if it does not exist. F_READ - open the file for reading only, return an error if the file does not exist. F_APPEND - open the file for reading and writing but does not truncate the file if exists; additionally, the file pointer references the end of the file.

Returns

file descriptor number or -1 on error

4.2.2.9 k_proc_create()

```
pcb_t* k_proc_create (
             pcb_t * parent,
             char * name )
```

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

Returns

Reference to the child PCB, or NULL if any error occurred.

4.2.2.10 k_proc_start()

```
void k_proc_start (
             pid_t pid )
```

Start a newly created process.

Parameters

pid	The process that will be started.

4.2.2.11 k_read()

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

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

Parameters

fd	the file descriptor num
n	number of bytes to read
buf	buffer to store bytes read

Returns

number of bytes read

4.2.2.12 k_set_foreground_pgid()

```
void k_set_foreground_pgid (
            pid_t pgid )
```

Set the foreground pgid, this is used for signaling.

Parameters

pid	the foreground pgid

4.2.2.13 k_sleep()

```
void k_sleep (
            unsigned int ticks )
```

Suspends execution of the calling proces for a specified number of clock ticks.

This function is analogous to `sleep(3)` in Linux, with the behavior that the system clock continues to tick even if the call is interrupted. The sleep can be interrupted by a P_SIGTERM signal, after which the function will return prematurely.

Parameters

ticks	Duration of the sleep in system clock ticks. Must be greater than 0.

4.2.2.14 k_start_kernel()

```
void k_start_kernel ( )
```

Start the kernel scheduler.

This function begins the process scheduling loop. It should be called after the kernel and initial processes (e.g., init) are set up.

4.2.2.15 k_waitpid()

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

Wait on a child of the calling process, until it changes state. If `nohang` is true, this will not block the calling process and return immediately.

Parameters

pid	Process ID of the child to wait for.
wstatus	Pointer to an integer variable where the status will be stored.
nohang	If true, return immediately if no child has exited.

Returns

pid_t The PCB of the child which has changed state on success, NULL on error.

4.2.2.16 k_write()

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

write n bytes of the string referenced by str to the file fd and increment the file pointer by n. On return, k_write returns the number of bytes written, or -1 on error.

Parameters

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

Returns

number of bytes written, or -1 if an error occurs

4.3 src/kernel/logging.h File Reference

Logging utility for kernel and process events.

Functions

- void init_logger (int logger_fd)

  Initialize the logger with a file descriptor.
- void log_event (char ∗msg)

  Log a single line event message.
- void logger_cleanup ()

  Clean up logger resources and close the file descriptor.

4.3.1 Detailed Description

Logging utility for kernel and process events.

Provides a simple interface for logging system events to a file descriptor. Used for debugging, auditing, or educational visualization of OS behavior.

4.3.2 Function Documentation

4.3.2.1 init_logger()

```
void init_logger (
            int logger_fd )
```

Initialize the logger with a file descriptor.

Parameters

logger↩ _fd	The file descriptor to which log messages will be written.

4.3.2.2 log_event()

```
void log_event (
            char * msg )
```

Log a single line event message.

Parameters

msg	Null-terminated string to write to the log.

4.4 src/kernel/p_signal.h File Reference

Defines process-level signal numbers used in the kernel.

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

Enumerations

- enum PSignalNum { P_SIGSTOP , P_SIGCONT , P_SIGTERM , P_SIGCHLD }

  Enumeration of supported process signals.

4.4.1 Detailed Description

Defines process-level signal numbers used in the kernel.

This header defines custom signal enumerations for inter-process communication and control, analogous to standard Unix signals.

4.4.2 Enumeration Type Documentation

4.4.2.1 PSignalNum

```
enum PSignalNum
```

Enumeration of supported process signals.

- P_SIGSTOP: Suspend the process (analogous to SIGSTOP).
- P_SIGCONT: Resume a suspended process (analogous to SIGCONT).
- P_SIGTERM: Terminate the process (analogous to SIGTERM).
- P_SIGCHLD: Sent to a parent when a child changes state (analogous to SIGCHLD).

Enumerator

P_SIGSTOP	Stop the process.
P_SIGCONT	Continue a stopped process.
P_SIGTERM	Terminate the process.
P_SIGCHLD	Notify parent of child status change.

4.5 src/kernel/pcb.h File Reference

Defines the Process Control Block (PCB) structure and related types.

```
#include "../util/spthread.h"
#include "../util/Vec.h"
```

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

Classes

- struct pcb

    The Process Control Block (PCB)

Typedefs

- typedef struct pcb **pcb_t**

Enumerations

- enum Status {
  CREATED , READY , SLEEP , WAITING ,
  STOPPED , ZOMBIE }

    Represents the runtime status of a process.
- enum Priority { HIGH , MID , LOW }

    Defines scheduling priority levels.
- enum ExitStatus {
  P_NOT_EXITED , P_EXITED , P_SIGNALED , P_CONTINUED ,
  P_STOPPED }

    Indicates how the process terminated or changed state.

Functions

- void pcb_destroy (void ∗pcb)

    Frees the memory used by a pcb_t object.

4.5.1 Detailed Description

Defines the Process Control Block (PCB) structure and related types.

The PCB is the core data structure for representing process state within the kernel. It contains scheduling information, status, file descriptors, thread info, and relationships to other processes.

4.5.2 Enumeration Type Documentation

4.5.2.1 ExitStatus

`enum ExitStatus`

Indicates how the process terminated or changed state.

Enumerator

P_NOT_EXITED	Still running.
P_EXITED	Exited normally.
P_SIGNALED	Terminated via signal.
P_CONTINUED	Continued after being stopped.
P_STOPPED	Currently stopped.

4.5.2.2 Priority

```
enum Priority
```

Defines scheduling priority levels.

Enumerator

HIGH	High priority.
MID	Medium priority.
LOW	Low priority.

4.5.2.3 Status

```
enum Status
```

Represents the runtime status of a process.

Enumerator

CREATED	Process has been created but not yet started.
READY	Ready to run or currently running.
SLEEP	Sleeping for a fixed number of clock ticks.
WAITING	Waiting on a child process.
STOPPED	Suspended via signal.
ZOMBIE	Exited but not yet reaped by parent.

4.5.3 Function Documentation

4.5.3.1 pcb_destroy()

```
void pcb_destroy (
            void * pcb )
```

Frees the memory used by a pcb_t object.

Parameters

pcb	Pointer to a pcb_t to destroy.

4.6 src/kernel/scheduler.h File Reference

Interface for the process scheduler in the kernel.

```
#include "../util/Vec.h"
#include "pcb.h"
```

Include dependency graph for scheduler.h:

Functions

- void init_scheduler ()

  Initialize scheduler. Should only been called once.
- void add_ready_process (pcb_t ∗pcb)

  Execute a process that is ready if one if available, do nothing otherwise.
- void remove_ready_process (pcb_t ∗pcb)

  Remove a process from ready queue. Noop if specified process does not exist.
- pcb_t ∗ select_process ()

  Select a process from ready queue.
- void update_priority (pid_t pid, Priority old_priority, Priority new_priority)

  Update the priority of a process that is in ready queue. If not already in queue, noop.
- void **scheduler_cleanup** ()

4.6.1 Detailed Description

Interface for the process scheduler in the kernel.

This header defines functions for managing the ready queues, selecting processes to execute based on priority, and updating scheduling state. The scheduler supports basic priority-based scheduling and process queue manipulation for kernel-level multitasking.

4.6.2 Function Documentation

4.6.2.1 add_ready_process()

```
void add_ready_process (
            pcb_t ∗ pcb )
```

Execute a process that is ready if one if available, do nothing otherwise.

Parameters

clock_tick	current clock tick, for logging purpose only

Returns

pid_t the pid of the process been executed, or -1

Add a new process to ready queue. If already in the queue, noop

4.6.2.2 select_process()

```
pcb_t* select_process ( )
```

Select a process from ready queue.

Returns

The pointer to the PCB of selected process or NULL if none is available.

4.6.2.3 update_priority()

```
void update_priority (
            pid_t pid,
            Priority old_priority,
            Priority new_priority )
```

Update the priority of a process that is in ready queue. If not already in queue, noop.

Precondition

The process should have been already added to the ready queue.

4.7 src/pennfat/fat_commands.h File Reference

File operation commands for PennFAT file system.

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

Include dependency graph for fat_commands.h:

Functions

- int update_file_entry_on_disk (file_system_t ∗fs, entry_location_t ∗entry_loc, file_entry_t ∗entry)
  Update a file entry on disk after modification.
- void touch_file (file_system_t ∗fs, const char ∗fname)
  Create a file or update its modification time.
- void mv (file_system_t ∗fs, const char ∗src, const char ∗dest)
  Rename or move a file in the file system.
- void rm (file_system_t ∗fs, const char ∗fname)
  Remove a file from the file system.
- void ∗ cat (void ∗arg)
  Concatenate files or redirect input/output (shell-level API) Usage: cat FILE1 [FILE2 ...] - Print content of files to stdout cat -w OUTPUT_FILE - Read from stdin and write to output file cat -a OUTPUT_FILE - Read from stdin and append to output file cat FILE1 FILE2 -w OUTPUT_FILE - Concatenate input files and write to output cat FILE1 -a OUTPUT_FILE - Concatenate input and append to output.
- void ∗ cat_fat_level (void ∗arg)
  Concatenate files or redirect input/output (direct kernel-level I/O)
- void cp (file_system_t ∗fs, int argc, char ∗argv[ ])
  Copy file between PennFAT and host system, or within PennFAT.
- void chmod_file (file_system_t ∗fs, const char ∗mode_str, const char ∗fname)
  Change permissions of a file.
- void ls_files (file_system_t ∗fs)
  List all files using kernel interface (high-level)
- void ls_files_fat_level (file_system_t ∗fs)
  List all files by scanning directory blocks (low-level)

4.7.1 Detailed Description

File operation commands for PennFAT file system.

This header defines user-facing file commands such as `touch`, `mv`, `rm`, `cat`, `cp`, `chmod`, and `ls`, which operate over the PennFAT virtual file system or bridge between host and PennFAT environments.

The implementation depends on `fat_kernel.h` for low-level file operations and supports permission management, content redirection, and file listing.

4.7.2 Function Documentation

4.7.2.1 cat()

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

Concatenate files or redirect input/output (shell-level API) Usage: cat FILE1 [FILE2 ...] - Print content of files to stdout cat -w OUTPUT_FILE - Read from stdin and write to output file cat -a OUTPUT_FILE - Read from stdin and append to output file cat FILE1 FILE2 -w OUTPUT_FILE - Concatenate input files and write to output cat FILE1 -a OUTPUT_FILE - Concatenate input and append to output.

Notes:

- When appending with `-a`, input and output files must not overlap.
- If `-w` is used, output file is truncated before writing.

Parameters

arg	Argument array (char∗[])

Returns

void∗ Always returns NULL

4.7.2.2 cat_fat_level()

```
void* cat_fat_level (
            void * arg )
```

Concatenate files or redirect input/output (direct kernel-level I/O)

Parameters

arg	Argument array (char∗[])

Returns

void∗ Always returns NULL

4.7.2.3 chmod_file()

```
void chmod_file (
            file_system_t * fs,
            const char * mode_str,
            const char * fname )
```

Change permissions of a file.

Parameters

fs	Pointer to the mounted file system
mode_str	Mode string in form "+rw", "-x", etc.
fname	Name of the file

4.7.2.4 cp()

```
void cp (
            file_system_t * fs,
            int argc,
            char * argv[] )
```

Copy file between PennFAT and host system, or within PennFAT.

Usage:

- cp -h HOST_SRC PENNFAT_DEST
- cp PENNFAT_SRC -h HOST_DEST
- cp PENNFAT_SRC PENNFAT_DEST

Parameters

fs	Pointer to the mounted file system
argc	Argument count
argv	Argument list

4.7.2.5 ls_files()

```
void ls_files (
              file_system_t * fs )
```

List all files using kernel interface (high-level)

Parameters

fs	Pointer to the mounted file system

4.7.2.6 ls_files_fat_level()

```
void ls_files_fat_level (
              file_system_t * fs )
```

List all files by scanning directory blocks (low-level)

Parameters

fs	Pointer to the mounted file system

4.7.2.7 mv()

```
void mv (
            file_system_t * fs,
            const char * src,
            const char * dest )
```

Rename or move a file in the file system.

Parameters

fs	Pointer to the mounted file system
src	Source filename
dest	Destination filename

4.7.2.8 rm()

```
void rm (
            file_system_t * fs,
            const char * fname )
```

Remove a file from the file system.

Parameters

fs	Pointer to the mounted file system
fname	Name of the file to remove

4.7.2.9 touch_file()

```
void touch_file (
            file_system_t * fs,
            const char * fname )
```

Create a file or update its modification time.

Parameters

fs	Pointer to the mounted file system
fname	Name of the file to create or update

4.7.2.10 update_file_entry_on_disk()

```
int update_file_entry_on_disk (
            file_system_t * fs,
            entry_location_t * entry_loc,
            file_entry_t * entry )
```

Update a file entry on disk after modification.

Parameters

fs	Pointer to the mounted file system
entry_loc	Location of the file entry on disk
entry	New file entry to be written

Returns

int 0 on success, -1 on failure

4.8 src/pennfat/fat_kernel.h File Reference

Kernel-level file operations for PennFAT file system.

```
#include "fat_tables.h"
#include "fat_utils.h"
#include "kernel/fd.h"
```

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

Macros

- #define **F_SEEK_SET** 0
- #define **F_SEEK_CUR** 1
- #define **F_SEEK_END** 2
- #define **K_ERROR_FILE_NOT_FOUND** -1
- #define **K_ERROR_FILE_ALREADY_OPEN** -2
- #define **K_ERROR_NO_FREE_DESCRIPTOR** -3
- #define **K_ERROR_INVALID_MODE** -4
- #define **K_ERROR_FILE_EXISTS** -5
- #define **K_ERROR_DIRECTORY_FULL** -6

Functions

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

  Open a file in the PennFAT file system.
- int fat_k_read (int fd, int n, char ∗buf)

  Read data from an open file.
- int fat_k_write (int fd, const char ∗str, int n)

  Write data to an open file.
- int fat_k_close (int fd)

Close an open file descriptor.
- int fat_k_lseek (int fd, int offset, int whence)

  Reposition the file offset.
- int fat_k_unlink (const char ∗fname)

  Delete a file from the file system.
- int fat_k_ls (const char ∗pattern)

  List information about files.
- file_system_t ∗ get_file_system ()

  Return a pointer to the currently mounted file system.

Variables

- file_system_t **current_fs**

4.8.1 Detailed Description

Kernel-level file operations for PennFAT file system.

This header exposes core system call–like interfaces for interacting with files in the PennFAT file system. Functions mimic standard UNIX-like behaviors (open, read, write, seek, close, unlink), and manage internal descriptor tables, FAT allocation, and file metadata.

4.8.2 Function Documentation

4.8.2.1 fat_k_close()

```
int fat_k_close (
            int fd )
```

Close an open file descriptor.

Releases system resources associated with the file descriptor.

Parameters

fd	File descriptor to close

Returns

0 on success, or negative error code

4.8.2.2 fat_k_ls()

```
int fat_k_ls (
            const char * pattern )
```

List information about files.

If `pattern` is NULL, lists all files. If a filename is provided, lists info for that specific file (if it exists).

Parameters

pattern	Filename pattern to match (or NULL for all files)

Returns

0 on success, or negative error code

4.8.2.3 fat_k_lseek()

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

Reposition the file offset.

Adjusts the current offset of the file descriptor based on the given `whence` mode: F_SEEK_SET (absolute), F_↩
SEEK_CUR (relative), or F_SEEK_END.

Parameters

fd	File descriptor
offset	Number of bytes to offset
whence	Seek mode (F_SEEK_SET, F_SEEK_CUR, F_SEEK_END)

Returns

New file offset on success, or negative error code

4.8.2.4 fat_k_open()

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

Open a file in the PennFAT file system.

Opens the file with the given name and mode. If the file does not exist and the mode implies creation (e.g., write/append), it will be created.

Parameters

fname	Filename to open
mode	File open mode (F_READ, F_WRITE, or F_APPEND)

Returns

File descriptor on success, or negative error code on failure

Open a file with specified mode

Parameters

fname	Filename to open
mode	Open mode (F_READ, F_WRITE, F_APPEND)

Returns

File descriptor on success, negative value on error

4.8.2.5 fat_k_read()

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

Read data from an open file.

Reads up to `n` bytes from the file associated with `fd` into the buffer `buf`. Reading past EOF returns 0 bytes.

Parameters

fd	File descriptor to read from
n	Maximum number of bytes to read
buf	Destination buffer

Returns

Number of bytes actually read, or negative error code

4.8.2.6 fat_k_unlink()

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

Delete a file from the file system.

Removes the file's entry and frees associated data blocks in the FAT. Fails if the file is currently open.

Parameters

fname	Filename to delete

Returns

0 on success, or negative error code

4.8.2.7 fat_k_write()

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

Write data to an open file.

Writes up to `n` bytes from the buffer `str` into the file associated with `fd`. Writing extends the file size and allocates new blocks if necessary.

Parameters

fd	File descriptor to write to
str	Source buffer
n	Number of bytes to write

Returns

Number of bytes actually written, or negative error code

4.8.2.8 get_file_system()

```
file_system_t* get_file_system ( )
```

Return a pointer to the currently mounted file system.

For use by the shell or upper layers that need access to the active file system's metadata, block size, FAT table, etc.

Returns

Pointer to the current file_system_t instance

4.9 src/pennfat/fat_mounting.h File Reference

Filesystem creation and mounting interface for PennFAT.

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

Functions

- int mkfs (const char ∗fs_name, uint16_t num_blocks, uint16_t block_size_config)

  Create a new PennFAT file system image.
- int mount_fs (const char ∗fs_name)

  Mount an existing PennFAT file system into memory.
- int unmount_fs (void)

  Unmount the currently mounted PennFAT file system.

4.9.1 Detailed Description

Filesystem creation and mounting interface for PennFAT.

This header provides functions to create, mount, and unmount a PennFAT file system. The mounting process maps the FAT image file into memory and initializes internal kernel structures for file access.

4.9.2 Function Documentation

4.9.2.1 mkfs()

```
int mkfs (
            const char * fs_name,
            uint16_t num_blocks,
            uint16_t block_size_config )
```

Create a new PennFAT file system image.

Initializes a blank file system file with valid headers, FAT table, and root directory block. This function should be used once to format a new FAT image.

Parameters

fs_name	Path to the file system image (e.g., "disk.pfat")
num_blocks	Total number of blocks (valid range: 1–32)
block_size_config	Block size configuration code: 0 = 128B, 1 = 256B, ..., 4 = 2048B

Returns

0 on success, -1 on error (e.g., invalid parameters or I/O failure)

Create a new PennFAT filesystem The structure of the filesystem is as follows:

- FAT table at the beginning
- Data blocks after the FAT table The first block of the data block is reserved for the root directory, where all the 64-byte file entries are stored. when the root directory is full, link to another block, and so on.

Parameters

fs_name	Name of the filesystem file
num_blocks	Number of blocks in FAT (1-32)
block_size_config	Block size configuration (0-4)

Returns

0 on success, -1 on error

4.9.2.2 mount_fs()

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

Mount an existing PennFAT file system into memory.

Loads the file system metadata, FAT table, and directory blocks into memory. This operation must be called before performing any file-level operations.

Parameters

fs_name	Path to an existing FAT image file

Returns

0 on success, -1 on failure (e.g., file not found or format invalid)

Mount a PennFAT filesystem Mainly reads the num_blocks of the FAT, block size, and mount the FAT into memory these parameters are stored in the global current_fs variable

Parameters

fs_name	Name of the filesystem file to mount

Returns

0 on success, -1 on error

4.9.2.3 unmount_fs()

```
int unmount_fs (
            void  )
```

Unmount the currently mounted PennFAT file system.

Flushes any changes to disk and releases memory mappings. After unmounting, file operations cannot proceed until another mount occurs.

Returns

0 on success, -1 on error (e.g., no file system was mounted)

Unmount the currently mounted filesystem

Returns

0 on success, -1 on error

4.10 src/pennfat/fat_tables.h File Reference

Core FAT data structures and constants for PennFAT.

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

Classes

- struct file_entry

    On-disk representation of a file entry (64 bytes total)
- struct entry_location

    Logical location of a file entry in a directory block.
- struct file_system

    In-memory structure for mounted file system metadata.
- struct file_descriptor_t

    In-memory metadata for open files (used by kernel)

Macros

- #define FAT_END_OF_CHAIN 0xFFFF

  Special FAT16 values.
- #define FAT_FREE_CLUSTER 0x0000

  Indicates an unused block.
- #define MAX_FILES 128

  File system layout limits.
- #define MAX_FILENAME_LENGTH 32

  Max file name length (bytes)
- #define MAX_FILE_DESCRIPTORS (16 ∗ 128)

  Max number of open files system-wide.
- #define FILE_TYPE_REGULAR 1

  File types.
- #define FILE_TYPE_DIRECTORY 2

  Directory file (not fully implemented)
- #define PERM_READ 0444

  File permission flags (Unix-like style)
- #define PERM_WRITE 0222

  Write-only permission.
- #define PERM_EXEC 0111

  Execute permission.
- #define PERM_RW 0666

  Read/write permission.
- #define PERM_RWX 0777

  Full permission.
- #define MODE_READ 0

  File open modes.
- #define MODE_WRITE 1

  Open file for writing (truncate or create)
- #define MODE_APPEND 2

  Open file for appending.
- #define SEEK_SET 0

  Seek modes for lseek-style navigation.
- #define SEEK_CUR 1

  Seek from current offset.
- #define SEEK_END 2

  Seek from end of file.

Typedefs

- typedef struct entry_location entry_location_t

  Logical location of a file entry in a directory block.
- typedef struct file_system file_system_t

  In-memory structure for mounted file system metadata.

Functions

- struct file_entry __attribute__ ((packed)) file_entry_t

  On-disk representation of a file entry (64 bytes total)

Variables

- char name [32]

  File name (null-terminated if shorter)
- uint32_t size

  File size in bytes.
- uint16_t firstBlock

  Index of first data block.
- uint8_t type

  FILE_TYPE_REGULAR or FILE_TYPE_DIRECTORY.
- uint8_t perm

  Permissions using PERM_∗ flags.
- time_t mtime

  Last modified timestamp.
- char reserved [16]

  Reserved for future use or padding.
- file_descriptor_t file_descriptor_table [MAX_FILE_DESCRIPTORS]

  Global table of file descriptors.

4.10.1 Detailed Description

Core FAT data structures and constants for PennFAT.

This header defines the fundamental constants, types, and data structures used in the PennFAT file system, including file entries, descriptor table, the FAT table, and in-memory file system metadata.

4.10.2 Macro Definition Documentation

4.10.2.1 FAT_END_OF_CHAIN

```
#define FAT_END_OF_CHAIN 0xFFFF
```

Special FAT16 values.

Marks end of a block chain

4.10.2.2 FILE_TYPE_REGULAR

```
#define FILE_TYPE_REGULAR 1
```

File types.

Regular data file

4.10.2.3 MAX_FILES

```
#define MAX_FILES 128
```

File system layout limits.

Max number of directory entries

4.10.2.4 MODE_READ

```
#define MODE_READ 0
```

File open modes.

Open file for reading

4.10.2.5 PERM_READ

```
#define PERM_READ 0444
```

File permission flags (Unix-like style)

Read-only permission

4.10.2.6 SEEK_SET

```
#define SEEK_SET 0
```

Seek modes for lseek-style navigation.

Seek from beginning of file

4.10.3 Typedef Documentation

4.10.3.1 entry_location_t

```
typedef struct entry_location entry_location_t
```

Logical location of a file entry in a directory block.

Used for locating or updating a specific file entry on disk.

4.10.3.2 file_system_t

```
typedef struct file_system file_system_t
```

In-memory structure for mounted file system metadata.

Represents the active PennFAT file system. Maintained in global memory after mounting.

4.10.4 Function Documentation

4.10.4.1 __attribute__()

```
struct file_entry __attribute__ (
            (packed)  )
```

On-disk representation of a file entry (64 bytes total)

Each file entry represents metadata for one file in the directory. All entries are stored packed into directory blocks.

4.11 src/pennfat/fat_utils.h File Reference

Low-level utilities for PennFAT block and file entry manipulation.

```
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <time.h>
#include "fat_tables.h"
```

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

Functions

- int read_block (uint16_t block_num, void ∗buffer)

  Read a block from disk into memory.
- int write_block (uint16_t block_num, const void ∗buffer)

  Write a memory buffer to a block on disk.
- uint16_t allocate_block (void)

  Allocate a free block in the FAT table.
- int sync_fat_table (void)

  Flush the in-memory FAT table to disk.
- int find_or_create_file_entry (const char ∗fname, entry_location_t ∗entry_loc, bool create, uint8_t type, uint16_t perm)

  Locate or create a file entry in the root directory.
- int get_file_entry (entry_location_t ∗entry_loc, file_entry_t ∗entry)

  Retrieve file entry metadata from a given location.
- int navigate_to_position (int fd, uint32_t target_offset)

  Navigate to a specific byte offset within a file.
- int update_file_size (int fd, uint32_t new_size)

  Update the file size in descriptor and disk.
- uint16_t allocate_file_block (int fd)

  Allocate a new block for a file.

4.11.1 Detailed Description

Low-level utilities for PennFAT block and file entry manipulation.

This header defines internal helper functions for block-level I/O, file entry management, FAT chain navigation, and file size tracking. These functions are not exposed to the user but are essential for correct kernel-level file system behavior.

4.11.2 Function Documentation

4.11.2.1 allocate_block()

```
uint16_t allocate_block (
            void )
```

Allocate a free block in the FAT table.

Finds the first available (free) cluster and marks it as end of chain.

Returns

Newly allocated block number, or 0 if none available

4.11.2.2 allocate_file_block()

```
uint16_t allocate_file_block (
            int fd )
```

Allocate a new block for a file.

Appends a new block to the file's FAT chain. If file is empty, sets the first block and updates the on-disk entry.

Parameters

fd	File descriptor

Returns

New block number on success, 0 on failure

4.11.2.3 find_or_create_file_entry()

```
int find_or_create_file_entry (
            const char * fname,
            entry_location_t * entry_loc,
            bool create,
            uint8_t type,
            uint16_t perm )
```

Locate or create a file entry in the root directory.

Searches for a file by name. If not found and `create` is true, allocates a new entry at the first free slot.

Parameters

fname	File name to find or create
entry_loc	Output location of the file entry
create	Whether to create a new entry if not found
type	File type (e.g., FILE_TYPE_REGULAR)
perm	File permissions

Returns

0 on success, -1 on error

4.11.2.4 get_file_entry()

```
int get_file_entry (
            entry_location_t * entry_loc,
            file_entry_t * entry )
```

Retrieve file entry metadata from a given location.

Reads the directory block and copies the file entry into memory.

Parameters

entry_loc	Location of the file entry
entry	Output pointer to populated file entry

Returns

0 on success, -1 on error

4.11.2.5 navigate_to_position()

```
int navigate_to_position (
            int fd,
            uint32_t target_offset )
```

Navigate to a specific byte offset within a file.

Traverses the FAT chain from the beginning or current position to reach the specified logical offset.

Parameters

fd	File descriptor index
target_offset	Desired file byte offset

Returns

0 on success, -1 on error

4.11.2.6 read_block()

```
int read_block (
            uint16_t block_num,
            void * buffer )
```

Read a block from disk into memory.

Parameters

block_num	Block number to read (starting from 1)
buffer	Destination buffer of size `block_size`

Returns

0 on success, -1 on error

4.11.2.7 sync_fat_table()

```
int sync_fat_table (
            void )
```

Flush the in-memory FAT table to disk.

Ensures consistency between memory and on-disk FAT copy using `msync`.

Returns

0 on success, -1 on error

4.11.2.8 update_file_size()

```
int update_file_size (
             int fd,
             uint32_t new_size )
```

Update the file size in descriptor and disk.

Adjusts in-memory and on-disk file size and update modification time.

Parameters

fd	File descriptor
new_size	New size in bytes

Returns

0 on success, -1 on error

4.11.2.9 write_block()

```
int write_block (
             uint16_t block_num,
             const void * buffer )
```

Write a memory buffer to a block on disk.

Parameters

block_num	Block number to write to
buffer	Source buffer of size `block_size`

Returns

0 on success, -1 on error

4.12 src/shell/jobs.h File Reference

Job control module for PennOS shell.

```
#include "parser.h"
```
Include dependency graph for jobs.h:

Classes

- struct job_st

Typedefs

- typedef struct job_st **job_t**

Functions

- void init_jobs ()

  Initialize job tracking system.
- void cleanup_jobs ()

  Clean up all job-related resources.
- void list_jobs (int output_fd)

  List all currently active or stopped jobs.
- char * concat_cmd (char **cmd)

  Concatenate a command array into a single string.
- int new_background_job (pid_t pid, char *cmd, char status)

  Create a new background job and add it to the job list.
- void check_job_status ()

  Check the status of all jobs and update internal records.
- void handle_bg (pid_t pgid)

  Resume a stopped job in the background.
- void handle_fg (pid_t pgid, char *cmd)

  Bring a job to the foreground and wait for its completion.
- void builtin_fg (void *arg)

  Shell built-in `fg` command.
- void builtin_bg (void *arg)

  Shell built-in `bg` command.
- int get_recent_job_id ()

  Get the job ID of the most recent job (running or stopped)
- job_t * get_job_ptr (pid_t pgid)

  Find a job by its PGID.
- job_t * get_job_by_id (int id)

  Find a job by its job ID.

4.12.1 Detailed Description

Job control module for PennOS shell.

This header defines the interface for managing background and foreground jobs using process group IDs (PGIDs). It supports job creation, tracking, status updates, and built-in shell commands like `fg` and `bg`.

4.12.2 Function Documentation

4.12.2.1 builtin_bg()

```
void builtin_bg (
            void * arg )
```

Shell built-in `bg` command.

Usage:

- bg → resumes most recent stopped job
- bg <job_id> → resumes job with specific ID

Parameters

arg	Argument list (char∗[]), passed from parser

4.12.2.2 builtin_fg()

```
void builtin_fg (
             void * arg )
```

Shell built-in `fg` command.

Usage:

- fg → resumes most recent job
- fg <job_id> → resumes job with specific ID

Parameters

arg	Argument list (char∗[]), passed from parser

4.12.2.3 check_job_status()

```
void check_job_status ( )
```

Check the status of all jobs and update internal records.

Prints job status transitions (e.g., Done, Killed, Stopped) to stdout.

4.12.2.4 cleanup_jobs()

```
void cleanup_jobs ( )
```

Clean up all job-related resources.

Called during shell shutdown.

4.12.2.5 concat_cmd()

```
char* concat_cmd (
             char ** cmd )
```

Concatenate a command array into a single string.

Useful for displaying commands in job output.

Parameters

cmd	Null-terminated argument list (char∗[])

Returns

Dynamically allocated string. Must be freed by caller.

4.12.2.6 get_job_by_id()

```
job_t* get_job_by_id (
            int id )
```

Find a job by its job ID.

Parameters

id	Internal job ID

Returns

Pointer to job_t struct, or NULL if not found

4.12.2.7 get_job_ptr()

```
job_t* get_job_ptr (
            pid_t pgid )
```

Find a job by its PGID.

Parameters

pgid	Process group ID

Returns

Pointer to job_t struct, or NULL if not found

4.12.2.8 get_recent_job_id()

```
int get_recent_job_id ( )
```

Get the job ID of the most recent job (running or stopped)

Returns

Job ID, or -1 if none

4.12.2.9 handle_bg()

```
void handle_bg (
            pid_t pgid )
```

Resume a stopped job in the background.

If the job is not stopped or not found, prints an error.

Parameters

pgid	Process group ID of the job

4.12.2.10 handle_fg()

```
void handle_fg (
            pid_t pgid,
            char * cmd )
```

Bring a job to the foreground and wait for its completion.

If the job stops (via SIGTSTP), it is added back as a background job.

Parameters

pgid	Process group ID of the job
cmd	Command string associated with the job

4.12.2.11 init_jobs()

```
void init_jobs ( )
```

Initialize job tracking system.

Must be called once at shell startup.

4.12.2.12 list_jobs()

```
void list_jobs (
            int output_fd )
```

List all currently active or stopped jobs.

Parameters

output←_fd	File descriptor to write the job list output to

4.12.2.13 new_background_job()

```
int new_background_job (
            pid_t pid,
            char * cmd,
            char status )
```

Create a new background job and add it to the job list.

Parameters

pid	Process group ID of the job
cmd	Full command string (assumed heap-allocated)
status	'r' for running or 's' for stopped

Returns

Job ID of the new job

4.13 src/shell/parser.h File Reference

Command-line parser for Penn-Shell.

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

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

Classes

- struct parsed_command

    Represents a parsed command line with optional I/O redirection, backgrounding, and pipelines.

Macros

- #define UNEXPECTED_FILE_INPUT 1

    Parser error: unexpected input redirection '<'.
- #define UNEXPECTED_FILE_OUTPUT 2

    Parser error: unexpected output redirection '>'.
- #define UNEXPECTED_PIPELINE 3

Parser error: unexpected pipeline '|'.
- #define UNEXPECTED_AMPERSAND 4

    Parser error: unexpected ampersand '&'.
- #define EXPECT_INPUT_FILENAME 5

    Parser error: missing filename after '<'.
- #define EXPECT_OUTPUT_FILENAME 6

    Parser error: missing filename after '>' or '>>'.
- #define EXPECT_COMMANDS 7

    Parser error: missing command or argument tokens.

Functions

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

    Parses a shell command line into a structured `parsed_command`
- void print_parsed_command (const struct parsed_command ∗cmd)

    Print a parsed command for debugging purposes.
- void print_parser_errcode (FILE ∗output, int err_code)

    Print a descriptive message corresponding to a parser error code.

4.13.1 Detailed Description

Command-line parser for Penn-Shell.

This header defines the `parsed_command` structure and the functions used to parse a shell input string into a structured representation for execution. It supports input/output redirection, pipelines, background execution, and provides detailed error reporting for invalid syntax.

4.13.2 Function Documentation

4.13.2.1 parse_command()

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

Parses a shell command line into a structured `parsed_command`

Parameters

cmd_line	Null-terminated string from user input
result	Output pointer to the parsed structure

Returns

- 0 on success (valid command parsed)

- -1 on system error (e.g., memory allocation failure)
- 1–7 on syntax error (see error code macros above)

Note

On success, the returned `*result` must be freed by the caller. On failure, `*result` is left unmodified.

layout of memory for `struct parsed_command` bool is_background; bool is_file_append;

const char *stdin_file; const char *stdout_file;

size_t num_commands;

commands are pointers to `arguments` char **commands[num_commands];

below are hidden in memory **

arguments are pointers to `original_string + num_commands` because all argv are null-terminated char *arguments[total_strings + num_commands];

original_string is a copy of the cmdline but with each token null-terminated char *original_string;

4.13.2.2 print_parsed_command()

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

Print a parsed command for debugging purposes.

Parameters

cmd	Parsed command structure (non-NULL)

4.13.2.3 print_parser_errcode()

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

Print a descriptive message corresponding to a parser error code.

Parameters

output	Output stream (e.g., stderr or a file)
err_code	One of the parser error codes (1–7)

4.14 src/shell/routines.h File Reference

Shell command routines for PennOS.

`#include <stdbool.h>`
Include dependency graph for routines.h:

Classes

- struct routine_args_st

Typedefs

- typedef void ∗(∗ spthread_fn) (void ∗)

  Function pointer type for shell routines.
- typedef struct routine_args_st **routine_args**

Functions

- void ∗ execute_routine (void ∗arg)

  Execute a command routine in a new thread.
- void ∗ busy (void ∗arg)

  Indefinitely busy loop.
- void ∗ echo (void ∗arg)

  Echo the arguments back to the standard output.
- void ∗ zombify (void ∗arg)

  Simulates zombified process behavior.
- void ∗ orphanify (void ∗arg)

  Simulates an orphaned process.
- void ∗ p_kill (void ∗arg)

  Sends signal(s) to target PID(s)
- void ∗ ps (void ∗arg)

  List all processes managed by PennOS.
- void ∗ p_sleep (void ∗arg)

  Sleep for N seconds (1 tick = 0.1s)
- void ∗ touch_wrapper (void ∗arg)

  Wrapper for `touch` command (FAT)
- void ∗ mv_wrapper (void ∗arg)

  Wrapper for `mv` command (FAT)
- void ∗ rm_wrapper (void ∗arg)

  Wrapper for `rm` command (FAT)
- void ∗ cp_wrapper (void ∗arg)

  Wrapper for `cp` command (FAT <-> host)
- void ∗ chmod_wrapper (void ∗arg)

  Wrapper for `chmod` command (FAT)
- void ∗ ls_wrapper (void ∗arg)

  Wrapper for `ls` command (FAT)
- void ∗ cat_wrapper (void ∗arg)

  Wrapper for `cat` command (FAT)

4.14.1 Detailed Description

Shell command routines for PennOS.

This header defines wrapper functions and built-in command handlers for PennOS, including `echo`, `ps`, `sleep`, `kill`, and FAT file system commands. These routines are executed as spthreads via the shell scheduler.

4.14.2 Function Documentation

4.14.2.1 busy()

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

Indefinitely busy loop.

Used to test preemption and signal handling.

Returns

Never returns.

Example: `busy`

4.14.2.2 cat_wrapper()

```
void* cat_wrapper (
            void * arg )
```

Wrapper for `cat` command (FAT)

Displays file contents or concatenates files. Supports redirection.

Example: `cat file1.txt file2.txt -w out.txt`

4.14.2.3 chmod_wrapper()

```
void* chmod_wrapper (
            void * arg )
```

Wrapper for `chmod` command (FAT)

Changes permission bits on a FAT file.

Example: `chmod +rw file.txt`

4.14.2.4 cp_wrapper()

```
void* cp_wrapper (
            void * arg )
```

Wrapper for `cp` command (FAT <-> host)

Supports copying from host to FAT, FAT to host, or FAT to FAT.

Example: `cp -h host.txt fat.txt` or `cp fat1.txt -h host_copy.txt`

4.14.2.5 echo()

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

Echo the arguments back to the standard output.

Parameters

arg	char* array of arguments

Returns

Always returns NULL.

Example: `echo Hello World`

4.14.2.6 execute_routine()

```
void* execute_routine (
            void * arg )
```

Execute a command routine in a new thread.

Automatically wraps foreground/background logic, priority setting, PGID assignment, and job registration.

4.14.2.7 ls_wrapper()

```
void* ls_wrapper (
            void * arg )
```

Wrapper for `ls` command (FAT)

Lists all files in the root directory of PennFAT.

Example: `ls`

4.14.2.8 mv_wrapper()

```
void* mv_wrapper (
            void * arg )
```

Wrapper for `mv` command (FAT)

Renames or moves a file within PennFAT.

Example: `mv old.txt new.txt`

4.14.2.9 orphanify()

```
void* orphanify (
            void * arg )
```

Simulates an orphaned process.

Parent exits after spawning, leaving child without a parent.

Example: `orphanify`

4.14.2.10 p_kill()

```
void* p_kill (
            void * arg )
```

Sends signal(s) to target PID(s)

If the first argument is a signal name (-term, -stop, -cont), it sends that signal. Otherwise, defaults to -term.

Example: `kill 1 2 3` or `kill -stop 42`

4.14.2.11 p_sleep()

```
void* p_sleep (
            void * arg )
```

Sleep for N seconds (1 tick = 0.1s)

Example: `sleep 5`

4.14.2.12 ps()

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

List all processes managed by PennOS.

Prints process ID, parent ID, priority, status, and command string.

Example: `ps`

4.14.2.13 rm_wrapper()

```
void* rm_wrapper (
            void * arg )
```

Wrapper for `rm` command (FAT)

Removes one or more files from PennFAT.

Example: `rm file1.txt file2.txt`

4.14.2.14 touch_wrapper()

```
void* touch_wrapper (
            void * arg )
```

Wrapper for `touch` command (FAT)

Creates or updates one or more files in PennFAT.

Example: `touch foo.txt bar.txt`

4.14.2.15 zombify()

```
void* zombify (
            void * arg )
```

Simulates zombified process behavior.

Spawns a child and exits without waiting. The child becomes a zombie.

Example: `zombify`

4.15 src/shell/shell.h File Reference

Entry point and main loop for the PennOS shell.

Functions

- void ∗ shell (void ∗arg)

  Shell main loop. Parses input and dispatches commands.

4.15.1 Detailed Description

Entry point and main loop for the PennOS shell.

Provides the main shell interface that parses input, handles built-in commands, manages foreground/background jobs, redirects I/O, and invokes routine execution.

4.15.2 Function Documentation

4.15.2.1 shell()

```
void* shell (
            void * arg )
```

Shell main loop. Parses input and dispatches commands.

Blocks SIGALRM, SIGINT, and SIGSTOP signals. Uses `getline` to read from stdin, invokes `parse_command`, then runs built-in commands or spawns routines.

Parameters

arg	Not used (can be NULL)

Returns

Always returns NULL on termination

4.16 src/shell/stress.h File Reference

Contains process stress-testing routines for PennOS.

Functions

- void ∗ hang (void ∗arg)

  Spawns 10 processes (`child_0` through `child_9`), waits on them with blocking. Intended to simulate heavy parallel process load.
- void ∗ nohang (void ∗arg)

  Same as `hang`, but uses non-blocking `s_waitpid` and polls.
- void ∗ recur (void ∗arg)

  Recursively spawns 26 generations (`Gen_A` through `Gen_Z`), waiting on each child.
- void ∗ crash (void ∗arg)

  Writes a patterned file and force-kills PennOS using SIGKILL. Used to test file system crash safety (e.g., fsync behavior).

4.16.1 Detailed Description

Contains process stress-testing routines for PennOS.

Includes recursive spawner, parallel spawner with blocking/non-blocking wait, and file system stress crash test.

4.16.2 Function Documentation

4.16.2.1 crash()

```
void* crash (
            void * arg )
```

Writes a patterned file and force-kills PennOS using SIGKILL. Used to test file system crash safety (e.g., fsync behavior).

Requires file system to support at least 5480 bytes.

Parameters

arg	Not used

Returns

NULL

4.16.2.2 hang()

```
void* hang (
            void * arg )
```

Spawns 10 processes (`child_0` through `child_9`), waits on them with blocking. Intended to simulate heavy parallel process load.

Parameters

arg	Not used

Returns

NULL

4.16.2.3 nohang()

```
void* nohang (
            void * arg )
```

Same as `hang`, but uses non-blocking `s_waitpid` and polls.

Parameters

arg	Not used

Returns

NULL

4.16.2.4 recur()

```
void* recur (
            void * arg )
```

Recursively spawns 26 generations (`Gen_A` through `Gen_Z`), waiting on each child.

Parameters

arg	Not used

Returns

NULL

4.17 src/sys_call/sys_call.h File Reference

User-level system call interface for PennOS.

```
#include <unistd.h>
#include "../kernel/fd.h"
#include "../kernel/p_signal.h"
#include "../util/Vec.h"
```

Include dependency graph for sys_call.h:

Functions

- pid_t s_spawn (void ∗(∗func)(void ∗), char ∗argv[ ], int fd0, int fd1)

    Create a child process that executes the function `func`. The child will retain some attributes of the parent.
- pid_t s_waitpid (pid_t pid, int ∗wstatus, bool nohang)

    Wait on a child of the calling process, until it changes state. If `nohang` is true, this will not block the calling process and return immediately.
- void s_sleep (unsigned int ticks)

    Suspends execution of the calling proces for a specified number of clock ticks.
- int s_kill (pid_t pid, int signal)

    Send a signal to a particular process.
- void s_exit ()

    Unconditionally exit the calling process.
- int s_nice (pid_t pid, int priority)

    Set the priority of the specified thread.
- int s_ps (Vec ∗vec)

    List all processes and print to output fd of current running process.
- void s_shutdown ()

    Logout penn os.
- int s_open (const char ∗fname, FileDescriptorMode mode)

    open a file name fname with the mode mode and return a file descriptor. The allowed modes are as follows:
- int s_write (int fd, const char ∗str, int n)
- int s_read (int fd, char ∗buf, int n)

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

    close the file fd and return 0 on success, or a negative value on failure.
- void s_set_foreground_pgid (pid_t pid)

    Set the foreground pgid, this is used for signaling.
- int s_setpgid (pid_t pid, pid_t pgid)

    Set the PGID of pid.

4.17.1 Detailed Description

User-level system call interface for PennOS.

This header defines the system call APIs exposed to user programs, providing safe and structured access to kernel functionality such as:

- Process creation (`s_spawn`), waiting (`s_waitpid`), termination (`s_exit`)
- Signal handling (`s_kill`, `s_setpgid`, `s_set_foreground_pgid`)
- Sleep and scheduling (`s_sleep`, `s_nice`)
- File operations (`s_open`, `s_write`, `s_read`, `s_close`, `s_unlink`)
- System monitoring (`s_ps`, `s_shutdown`)

These functions serve as the user-facing interface to PennOS kernel internals, enabling user-level threads to interact with the kernel through controlled APIs.

Each `s_*` function wraps and invokes its corresponding kernel call (`k_*`), allowing for encapsulated resource management and process isolation.

Author

PennOS GROUP 0

Date

2025

4.17.2 Function Documentation

4.17.2.1 s_close()

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

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

Parameters

fd	the file descriptor num

Returns

0 on success, -1 on error

4.17.2.2 s_kill()

```
int s_kill (
            pid_t pid,
            int signal )
```

Send a signal to a particular process.

Parameters

pid	Process ID of the target proces.
signal	Signal number to be sent.

Returns

0 on success, -1 on error.

4.17.2.3 s_nice()

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

Set the priority of the specified thread.

Parameters

pid	Process ID of the target thread.
priority	The new priority value of the thread (0, 1, or 2)

Returns

0 on success, -1 on failure.

4.17.2.4 s_open()

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

open a file name fname with the mode mode and return a file descriptor. The allowed modes are as follows:

Parameters

fname	name of the file
mode	F_WRITE, writing and reading, truncates if the file exists, or creates it if it does not exist. F_READ open the file for reading only, return an error if the file does not exist. F_APPEND open the file for reading and writing but does not truncate the file if exists; additionally, the file pointer references the end of the file.

Returns

file descriptor number to use, or -1 on error

4.17.2.5 s_ps()

```
int s_ps (
            Vec * vec )
```

List all processes and print to output fd of current running process.

Parameters

vec	A vec to store the info

Returns

0 on success, -1 on error.

4.17.2.6 s_read()

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

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

Parameters

fd	the file descriptor num
n	number of bytes to read
buf	buffer to store bytes read

Returns

number of bytes read

4.17.2.7 s_set_foreground_pgid()

```
void s_set_foreground_pgid (
            pid_t pid )
```

Set the foreground pgid, this is used for signaling.

Parameters

pid	the foreground pgid

4.17.2.8 s_sleep()

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

Suspends execution of the calling proces for a specified number of clock ticks.

This function is analogous to `sleep(3)` in Linux, with the behavior that the system clock continues to tick even if the call is interrupted. The sleep can be interrupted by a P_SIGTERM signal, after which the function will return prematurely.

Parameters

ticks	Duration of the sleep in system clock ticks. Must be greater than 0.

4.17.2.9 s_spawn()

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

Create a child process that executes the function `func`. The child will retain some attributes of the parent.

Parameters

func	Function to be executed by the child process.
argv	Null-terminated array of args, including the command name as argv[0].
fd0	Input file descriptor.
fd1	Output file descriptor.

Returns

pid_t The process ID of the created child process.

4.17.2.10 s_waitpid()

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

Wait on a child of the calling process, until it changes state. If `nohang` is true, this will not block the calling process and return immediately.

Parameters

pid	Process ID of the child to wait for.
wstatus	Pointer to an integer variable where the status will be stored.
nohang	If true, return immediately if no child has exited.

Returns

pid_t The process ID of the child which has changed state on success, -1 on error.

4.17.2.11 s_write()

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

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

Parameters

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

Returns

number of bytes written

4.18 src/user/user.h File Reference

User-level utilities and macros for interacting with PennOS system calls.

```
#include <stdbool.h>
#include "../kernel/error.h"
```

Include dependency graph for user.h:

Macros

- #define **P_STD_IN** 0
- #define **P_STD_OUT** 1
- #define **P_STD_ERR** 2

Functions

- bool P_WIFCONTINUED (int status)

  Check if a process was resumed after being stopped.
- bool P_WIFEXITED (int status)

  Check if a process exited normally.
- bool P_WIFSTOPPED (int status)

  Check if a process is currently stopped.
- bool P_WIFSIGNALED (int status)

  Check if a process was terminated by a signal.
- void u_perror (const char *msg)

  Print a system error message corresponding to the last kernel error.
- bool u_write (int fd, const char *msg)

  Write the entire message to the given file descriptor.

4.18.1 Detailed Description

User-level utilities and macros for interacting with PennOS system calls.

This header defines user-space helper functions and macros:

- Macros for checking process status (e.g., WIFEXITED)
- Standard file descriptor constants (P_STD_IN, P_STD_OUT, P_STD_ERR)
- Error printing and safe writing helpers

4.18.2 Function Documentation

4.18.2.1 P_WIFCONTINUED()

```
bool P_WIFCONTINUED (
             int status )
```

Check if a process was resumed after being stopped.

Parameters

status	Process status code

Returns

true if continued, false otherwise

4.18.2.2 P_WIFEXITED()

```
bool P_WIFEXITED (
             int status )
```

Check if a process exited normally.

Parameters

status	Process status code

Returns

true if exited, false otherwise

4.18.2.3 P_WIFSIGNALED()

```
bool P_WIFSIGNALED (
             int status )
```

Check if a process was terminated by a signal.

Parameters

status	Process status code

Returns

true if signaled, false otherwise

4.18.2.4 P_WIFSTOPPED()

```
bool P_WIFSTOPPED (
             int status )
```

Check if a process is currently stopped.

Parameters

status	Process status code

Returns

true if stopped, false otherwise

4.18.2.5 u_perror()

```
void u_perror (
             const char * msg )
```

Print a system error message corresponding to the last kernel error.

Parameters

msg	User-provided context string

4.18.2.6 u_write()

```
bool u_write (
             int fd,
             const char * msg )
```

Write the entire message to the given file descriptor.

Parameters

fd	File descriptor
msg	Null-terminated string to write

Returns

true on success, false on failure

4.19 src/util/interface.h File Reference

Process information interface for PennOS process inspection.

`#include <unistd.h>`
Include dependency graph for interface.h:

Classes

- struct process_info

Typedefs

- typedef struct process_info **process_info_t**

Functions

- void free_process_info (void ∗ptr)

    Frees memory allocated for a process_info_t structure.

4.19.1 Detailed Description

Process information interface for PennOS process inspection.

Defines the structure for holding process metadata (PID, status, priority, etc.) and helper functions for managing dynamic memory associated with it.

4.19.2 Function Documentation

4.19.2.1 free_process_info()

```
void free_process_info (
            void * ptr )
```

Frees memory allocated for a process_info_t structure.

Parameters

ptr	Pointer to the process_info_t to free.

Index