PennOS

Generated by Doxygen 1.9.1

1 Class Index	1
1.1 Class List	. 1
2 File Index	3
2.1 File List	. 3
3 Class Documentation	5
3.1 dir_entry_t Struct Reference	. 5
3.1.1 Detailed Description	. 5
3.1.2 Member Data Documentation	. 5
3.1.2.1 firstBlock	. 5
3.1.2.2 mtime	. 6
3.1.2.3 name	. 6
3.1.2.4 perm	. 6
3.1.2.5 reserved	. 6
3.1.2.6 size	. 6
3.1.2.7 type	. 6
3.2 fd_entry_t Struct Reference	. 7
3.2.1 Detailed Description	. 7
3.2.2 Member Data Documentation	. 7
3.2.2.1 filename	. 7
3.2.2.2 first_block	. 7
3.2.2.3 in_use	. 7
3.2.2.4 mode	. 8
3.2.2.5 position	. 8
3.2.2.6 ref_count	. 8
3.2.2.7 size	. 8
3.3 job_st Struct Reference	. 8
3.3.1 Detailed Description	. 9
3.3.2 Member Data Documentation	. 9
3.3.2.1 cmd	. 9
3.3.2.2 finished_count	. 9
3.3.2.3 id	. 9
3.3.2.4 num_pids	. 9
3.3.2.5 pgid	. 10
3.3.2.6 pids	. 10
3.3.2.7 state	. 10
3.4 parsed_command Struct Reference	. 10
3.4.1 Detailed Description	. 10
3.4.2 Member Data Documentation	
3.4.2.1 commands	
3.4.2.2 is_background	. 11
3.4.2.3 is_file_append	

3.4.2.4 num_commands	 . 11
3.4.2.5 stdin_file	 . 11
3.4.2.6 stdout_file	 . 12
3.5 pcb_st Struct Reference	 . 12
3.5.1 Detailed Description	 . 13
3.5.2 Member Data Documentation	 . 13
3.5.2.1 child_pcbs	 . 13
3.5.2.2 cmd_str	 . 13
3.5.2.3 fd_table	 . 13
3.5.2.4 input_fd	 . 13
3.5.2.5 is_sleeping	 . 14
3.5.2.6 output_fd	 . 14
3.5.2.7 par_pid	 . 14
3.5.2.8 pid	 . 14
3.5.2.9 priority	 . 14
3.5.2.10 process_state	 . 14
3.5.2.11 process_status	 . 15
3.5.2.12 signals	 . 15
3.5.2.13 thread_handle	 . 15
3.5.2.14 time_to_wake	 . 15
3.6 spthread_fwd_args_st Struct Reference	 . 15
3.6.1 Detailed Description	 . 16
3.6.2 Member Data Documentation	 . 16
3.6.2.1 actual_arg	 . 16
3.6.2.2 actual_routine	 . 16
3.6.2.3 child_meta	 . 16
3.6.2.4 setup_cond	 . 16
3.6.2.5 setup_done	 . 17
3.6.2.6 setup_mutex	 . 17
3.7 spthread_meta_st Struct Reference	 . 17
3.7.1 Detailed Description	 . 17
3.7.2 Member Data Documentation	 . 17
3.7.2.1 meta_mutex	 . 17
3.7.2.2 state	 . 18
3.7.2.3 suspend_set	 . 18
3.8 spthread_signal_args_st Struct Reference	 . 18
3.8.1 Detailed Description	 . 18
3.8.2 Member Data Documentation	 . 18
3.8.2.1 ack	 . 18
3.8.2.2 shutup_mutex	 . 19
3.8.2.3 signal	 . 19
3.9 spthread_st Struct Reference	 . 19

3.9.1 Detailed Description	 . 19
3.9.2 Member Data Documentation	 . 20
3.9.2.1 meta	 . 20
3.9.2.2 thread	 . 20
3.10 vec_st Struct Reference	 . 20
3.10.1 Detailed Description	 . 20
3.10.2 Member Data Documentation	 . 20
3.10.2.1 capacity	 . 21
3.10.2.2 data	 . 21
3.10.2.3 ele_dtor_fn	 . 21
3.10.2.4 length	 . 21
4 File Documentation	23
4.1 SRC/fs/fat_routines.c File Reference	 . 23
4.1.1 Function Documentation	 . 24
4.1.1.1 cat()	 . 24
4.1.1.2 chmod()	 . 27
4.1.1.3 cmpctdir()	 . 28
4.1.1.4 cp()	 . 28
4.1.1.5 ls()	 . 29
4.1.1.6 mkfs()	 . 30
4.1.1.7 mount()	 . 32
4.1.1.8 mv()	 . 32
4.1.1.9 rm()	 . 34
4.1.1.10 touch()	 . 35
4.1.1.11 unmount()	 . 36
4.2 SRC/fs/fat_routines.h File Reference	 . 36
4.2.1 Macro Definition Documentation	 . 38
4.2.1.1 F_APPEND	 . 38
4.2.1.2 F_READ	 . 38
4.2.1.3 F_WRITE	 . 39
4.2.1.4 FAT_EOF	 . 39
4.2.1.5 FAT_FREE	 . 39
4.2.1.6 PERM_EXEC	 . 39
4.2.1.7 PERM_NONE	 . 39
4.2.1.8 PERM_READ	 . 39
4.2.1.9 PERM_READ_EXEC	 . 40
4.2.1.10 PERM_READ_WRITE	 . 40
4.2.1.11 PERM_READ_WRITE_EXEC	 . 40
4.2.1.12 PERM_WRITE	 . 40
4.2.1.13 TYPE_DIRECTORY	 . 40
4.2.1.14 TYPE_REGULAR	 . 40

4.2.1.15 TYPE_SYMLINK	. 41
4.2.1.16 TYPE_UNKNOWN	. 41
4.2.2 Function Documentation	. 41
4.2.2.1 cat()	. 41
4.2.2.2 chmod()	. 44
4.2.2.3 cmpctdir()	. 45
4.2.2.4 cp()	. 46
4.2.2.5 ls()	. 47
4.2.2.6 mkfs()	. 48
4.2.2.7 mount()	. 50
4.2.2.8 mv()	. 51
4.2.2.9 rm()	. 52
4.2.2.10 touch()	. 53
4.2.2.11 unmount()	. 55
4.3 SRC/fs/fs_helpers.c File Reference	. 55
4.3.1 Function Documentation	. 57
4.3.1.1 add_file_entry()	. 57
4.3.1.2 allocate_block()	. 58
4.3.1.3 compact_directory()	. 59
4.3.1.4 copy_host_to_pennfat()	. 61
4.3.1.5 copy_pennfat_to_host()	. 63
4.3.1.6 copy_source_to_dest()	. 64
4.3.1.7 decrement_fd_ref_count()	. 65
4.3.1.8 find_file()	. 66
4.3.1.9 get_free_fd()	. 67
4.3.1.10 has_executable_permission()	. 67
4.3.1.11 increment_fd_ref_count()	. 67
4.3.1.12 init_fd_table()	. 68
4.3.1.13 mark_entry_as_deleted()	. 69
4.3.2 Variable Documentation	. 69
4.3.2.1 block_size	. 69
4.3.2.2 fat	. 69
4.3.2.3 fat_size	. 70
4.3.2.4 fd_table	. 70
4.3.2.5 fs_fd	. 70
4.3.2.6 is_mounted	. 70
4.3.2.7 MAX_FDS	. 70
4.3.2.8 num_fat_blocks	. 70
4.4 SRC/fs/fs_helpers.h File Reference	. 71
4.4.1 Function Documentation	. 72
4.4.1.1 add_file_entry()	. 72
4.4.1.2 allocate_block()	. 74

4.4.1.3 compact_directory()	 75
4.4.1.4 copy_host_to_pennfat()	 77
4.4.1.5 copy_pennfat_to_host()	 79
4.4.1.6 copy_source_to_dest()	 80
4.4.1.7 decrement_fd_ref_count()	 81
4.4.1.8 find_file()	 82
4.4.1.9 get_free_fd()	 83
4.4.1.10 has_executable_permission()	 84
4.4.1.11 increment_fd_ref_count()	 85
4.4.1.12 init_fd_table()	 85
4.4.1.13 mark_entry_as_deleted()	 86
4.4.2 Variable Documentation	 87
4.4.2.1 block_size	 87
4.4.2.2 fat	 87
4.4.2.3 fat_size	 87
4.4.2.4 fd_table	 87
4.4.2.5 fs_fd	 88
4.4.2.6 is_mounted	 88
4.4.2.7 MAX_FDS	 88
4.4.2.8 num_fat_blocks	 88
4.5 SRC/fs/fs_kfuncs.c File Reference	 88
4.5.1 Function Documentation	 89
4.5.1.1 format_file_info()	 89
4.5.1.2 k_close()	 90
4.5.1.3 k_ls()	 90
4.5.1.4 k_lseek()	 91
4.5.1.5 k_open()	 92
4.5.1.6 k_read()	 94
4.5.1.7 k_unlink()	 95
4.5.1.8 k_write()	 96
4.5.2 Variable Documentation	 99
4.5.2.1 current_fg_pid	 99
4.5.2.2 current_running_pcb	 100
4.6 SRC/fs/fs_kfuncs.h File Reference	 100
4.6.1 Macro Definition Documentation	 101
4.6.1.1 SEEK_CUR	 101
4.6.1.2 SEEK_END	 101
4.6.1.3 SEEK_SET	 101
4.6.2 Function Documentation	 101
4.6.2.1 k_close()	 101
4.6.2.2 k_ls()	 102
4.6.2.3 k_lseek()	 104

4.6.2.4 k_open()	. 105
4.6.2.5 k_read()	. 107
4.6.2.6 k_unlink()	. 109
4.6.2.7 k_write()	. 110
4.7 SRC/fs/fs_syscalls.c File Reference	. 113
4.7.1 Function Documentation	. 114
4.7.1.1 s_close()	. 115
4.7.1.2 s_ls()	. 115
4.7.1.3 s_lseek()	. 115
4.7.1.4 s_open()	. 116
4.7.1.5 s_read()	. 116
4.7.1.6 s_unlink()	. 116
4.7.1.7 s_write()	
4.8 SRC/fs/fs_syscalls.h File Reference	. 117
4.8.1 Macro Definition Documentation	. 118
4.8.1.1 STDERR_FILENO	. 118
4.8.1.2 STDIN_FILENO	. 118
4.8.1.3 STDOUT_FILENO	. 118
4.8.2 Function Documentation	. 118
4.8.2.1 s_close()	. 118
4.8.2.2 s_ls()	
4.8.2.3 s_lseek()	. 120
4.8.2.4 s_open()	. 120
4.8.2.5 s_read()	. 121
4.8.2.6 s_unlink()	. 122
4.8.2.7 s_write()	. 122
4.9 SRC/kernel/kern_pcb.c File Reference	. 123
4.9.1 Function Documentation	. 124
4.9.1.1 create_pcb()	. 124
4.9.1.2 free_pcb()	. 125
4.9.1.3 k_proc_cleanup()	. 125
4.9.1.4 k_proc_create()	. 126
4.9.1.5 remove_child_in_parent()	. 127
4.9.2 Variable Documentation	. 127
4.9.2.1 current_pcbs	. 127
4.9.2.2 current_running_pcb	. 127
4.9.2.3 next_pid	. 127
4.10 SRC/kernel/kern_pcb.h File Reference	. 128
4.10.1 Macro Definition Documentation	. 129
4.10.1.1 FILE_DESCRIPTOR_TABLE_SIZE	
4.10.2 Typedef Documentation	. 129
4.10.2.1 pcb_t	. 129

4.10.3 Function Documentation	129
4.10.3.1 create_pcb()	129
4.10.3.2 free_pcb()	130
4.10.3.3 k_proc_cleanup()	131
4.10.3.4 k_proc_create()	132
4.10.3.5 remove_child_in_parent()	133
4.11 SRC/kernel/kern_sys_calls.c File Reference	133
4.11.1 Function Documentation	135
4.11.1.1 delete_from_explicit_queue()	135
4.11.1.2 delete_from_queue()	135
4.11.1.3 determine_index_in_queue()	136
4.11.1.4 init_func()	136
4.11.1.5 move_pcb_correct_queue()	136
4.11.1.6 s_cleanup_init_process()	137
4.11.1.7 s_echo()	137
4.11.1.8 s_exit()	138
4.11.1.9 s_kill()	138
4.11.1.10 s_nice()	
4.11.1.11 s_ps()	139
4.11.1.12 s_sleep()	140
4.11.1.13 s_spawn()	140
4.11.1.14 s_spawn_init()	141
4.11.1.15 s_waitpid()	141
4.11.2 Variable Documentation	142
4.11.2.1 current_fg_pid	142
4.11.2.2 current_pcbs	142
4.11.2.3 current_running_pcb	143
4.11.2.4 one_priority_queue	143
4.11.2.5 sleep_blocked_queue	143
4.11.2.6 tick_counter	143
4.11.2.7 two_priority_queue	143
4.11.2.8 zero_priority_queue	143
4.11.2.9 zombie_queue	144
4.12 SRC/kernel/kern_sys_calls.h File Reference	144
4.12.1 Function Documentation	145
4.12.1.1 delete_from_explicit_queue()	145
4.12.1.2 delete_from_queue()	
4.12.1.3 determine_index_in_queue()	146
4.12.1.4 init_func()	147
4.12.1.5 move_pcb_correct_queue()	
4.12.1.6 s_cleanup_init_process()	148
4.12.1.7 s_echo()	149

4.12.1.8 s_exit()	49
4.12.1.9 s_kill()	50
4.12.1.10 s_nice()	51
4.12.1.11 s_ps()	52
4.12.1.12 s_sleep()	52
4.12.1.13 s_spawn()	53
4.12.1.14 s_spawn_init()	54
4.12.1.15 s_waitpid()	54
4.13 SRC/kernel/logger.c File Reference	55
4.13.1 Function Documentation	56
4.13.1.1 log_generic_event()	56
4.13.1.2 log_nice_event()	57
4.13.1.3 log_scheduling_event()	58
4.14 SRC/kernel/logger.h File Reference	58
4.14.1 Function Documentation	59
4.14.1.1 log_generic_event()	59
4.14.1.2 log_nice_event()	60
4.14.1.3 log_scheduling_event()	61
4.14.2 Variable Documentation	61
4.14.2.1 log_fd	61
4.14.2.2 tick_counter	62
4.15 SRC/kernel/scheduler.c File Reference	62
4.15.1 Function Documentation	63
4.15.1.1 alarm_handler()	63
4.15.1.2 child_in_zombie_queue()	64
4.15.1.3 child_with_changed_process_status()	64
4.15.1.4 delete_process_from_all_queues()	64
4.15.1.5 delete_process_from_all_queues_except_current()	65
4.15.1.6 delete_process_from_particular_queue()	65
4.15.1.7 free_scheduler_queues()	65
4.15.1.8 generate_next_priority()	66
4.15.1.9 get_next_pcb()	66
4.15.1.10 get_pcb_in_queue()	67
4.15.1.11 handle_signal()	67
4.15.1.12 initialize_scheduler_queues()	68
4.15.1.13 put_pcb_into_correct_queue()	68
4.15.1.14 s_shutdown_pennos()	68
4.15.1.15 scheduler()	69
4.15.2 Variable Documentation	70
4.15.2.1 curr_priority_arr_index	70
4.15.2.2 current_pcbs	70
4.15.2.3 current_running_pcb	71

171
171
171
171
171
172
172
172
172
173
173
174
174
175
175
176
176
177
177
178
179
179
180
180
181
182
183
183
183
183
184
184
184
184
184
185
185
185
185
186
186
186
1111111111111111111111111111111

4.19.1.3 nohang()
4.19.1.4 recur()
4.20 SRC/kernel/stress.h File Reference
4.20.1 Function Documentation
4.20.1.1 crash()
4.20.1.2 hang()
4.20.1.3 nohang()
4.20.1.4 recur()
4.21 SRC/lib/pennos-errno.c File Reference
4.21.1 Variable Documentation
4.21.1.1 P_ERRNO
4.22 SRC/lib/pennos-errno.h File Reference
4.22.1 Macro Definition Documentation
4.22.1.1 P_EBADF
4.22.1.2 P_EBUSY
4.22.1.3 P_ECLOSE
4.22.1.4 P_ECOMMAND
4.22.1.5 P_EEXIST
4.22.1.6 P_EFS_NOT_MOUNTED
4.22.1.7 P_EFULL
4.22.1.8 P_EFUNC
4.22.1.9 P_EINTR
4.22.1.10 P_EINVAL
4.22.1.11 P_ELSEEK
4.22.1.12 P_EMALLOC
4.22.1.13 P_EMAP
4.22.1.14 P_ENOENT
4.22.1.15 P_ENULL
4.22.1.16 P_EOPEN
4.22.1.17 P_EPARSE
4.22.1.18 P_EPERM
4.22.1.19 P_EREAD
4.22.1.20 P_ESIGNAL
4.22.1.21 P_EUNKNOWN
4.22.1.22 P_EWRITE
4.22.1.23 P_INITFAIL
4.22.1.24 P_NEEDF
4.22.2 Variable Documentation
4.22.2.1 P_ERRNO
4.23 SRC/lib/spthread.c File Reference
4.23.1 Macro Definition Documentation
4.23.1.1 _GNU_SOURCE

4.23.1.2 MILISEC_IN_NANO	195
4.23.1.3 SPTHREAD_RUNNING_STATE	196
4.23.1.4 SPTHREAD_SIG_CONTINUE	196
4.23.1.5 SPTHREAD_SIG_SUSPEND	196
4.23.1.6 SPTHREAD_SUSPENDED_STATE	196
4.23.1.7 SPTHREAD_TERMINATED_STATE	196
4.23.2 Typedef Documentation	196
4.23.2.1 pthread_fn	196
4.23.2.2 spthread_fwd_args	197
4.23.2.3 spthread_meta_t	197
4.23.2.4 spthread_signal_args	197
4.23.3 Function Documentation	197
4.23.3.1 spthread_cancel()	197
4.23.3.2 spthread_continue()	197
4.23.3.3 spthread_create()	198
4.23.3.4 spthread_disable_interrupts_self()	199
4.23.3.5 spthread_enable_interrupts_self()	199
4.23.3.6 spthread_equal()	200
4.23.3.7 spthread_exit()	200
4.23.3.8 spthread_join()	200
4.23.3.9 spthread_self()	200
4.23.3.10 spthread_suspend()	201
4.23.3.11 spthread_suspend_self()	201
4.24 SRC/lib/spthread.h File Reference	202
4.24.1 Macro Definition Documentation	203
4.24.1.1 SIGPTHD	203
4.24.2 Typedef Documentation	203
4.24.2.1 spthread_meta_t	203
4.24.2.2 spthread_t	203
4.24.3 Function Documentation	203
4.24.3.1 spthread_cancel()	204
4.24.3.2 spthread_continue()	204
4.24.3.3 spthread_create()	205
4.24.3.4 spthread_disable_interrupts_self()	205
4.24.3.5 spthread_enable_interrupts_self()	206
4.24.3.6 spthread_equal()	206
4.24.3.7 spthread_exit()	206
4.24.3.8 spthread_join()	207
4.24.3.9 spthread_self()	207
4.24.3.10 spthread_suspend()	207
4.24.3.11 spthread_suspend_self()	208
4.25 SRC/lib/Vec.c File Reference	208

4.25.1 Function Documentation
4.25.1.1 vec_clear()
4.25.1.2 vec_destroy()
4.25.1.3 vec_erase()
4.25.1.4 vec_erase_no_deletor()
4.25.1.5 vec_get()
4.25.1.6 vec_insert()
4.25.1.7 vec_new()
4.25.1.8 vec_pop_back()
4.25.1.9 vec_push_back()
4.25.1.10 vec_resize()
4.25.1.11 vec_set()
4.26 SRC/lib/Vec.h File Reference
4.26.1 Macro Definition Documentation
4.26.1.1 vec_capacity
4.26.1.2 vec_is_empty
4.26.1.3 vec_len
4.26.2 Typedef Documentation
4.26.2.1 ptr_dtor_fn
4.26.2.2 ptr_t
4.26.2.3 Vec
4.26.3 Function Documentation
4.26.3.1 vec_clear()
4.26.3.2 vec_destroy()
4.26.3.3 vec_erase()
4.26.3.4 vec_erase_no_deletor()
4.26.3.5 vec_get()
4.26.3.6 vec_insert()
4.26.3.7 vec_new()
4.26.3.8 vec_pop_back()
4.26.3.9 vec_push_back()
4.26.3.10 vec_resize()
4.26.3.11 vec_set()
4.27 SRC/pennfat.c File Reference
4.27.1 Macro Definition Documentation
4.27.1.1 PROMPT
4.27.2 Function Documentation
4.27.2.1 main()
4.28 SRC/pennos.c File Reference
4.28.1 Function Documentation
4.28.1.1 main()
4.28.2 Variable Documentation

4.28.2.1 log_fd
4.28.2.2 tick_counter
4.29 SRC/shell/builtins.c File Reference
4.29.1 Function Documentation
4.29.1.1 u_perror()
4.30 SRC/shell/builtins.h File Reference
4.30.1 Function Documentation
4.30.1.1 u_perror()
4.31 SRC/shell/Job.h File Reference
4.31.1 Typedef Documentation
4.31.1.1 jid_t
4.31.1.2 job
4.31.2 Enumeration Type Documentation
4.31.2.1 job_state_t
4.32 SRC/shell/parser.c File Reference
4.32.1 Macro Definition Documentation
4.32.1.1 JUMP_OUT
4.32.2 Function Documentation
4.32.2.1 parse_command()
4.32.2.2 print_parsed_command()
4.32.2.3 print_parser_errcode()
4.33 SRC/shell/parser.h File Reference
4.33.1 Macro Definition Documentation
4.33.1.1 EXPECT_COMMANDS
4.33.1.2 EXPECT_INPUT_FILENAME
4.33.1.3 EXPECT_OUTPUT_FILENAME
4.33.1.4 UNEXPECTED_AMPERSAND
4.33.1.5 UNEXPECTED_FILE_INPUT
4.33.1.6 UNEXPECTED_FILE_OUTPUT
4.33.1.7 UNEXPECTED_PIPELINE
4.33.2 Function Documentation
4.33.2.1 parse_command()
4.33.2.2 print_parsed_command()
4.33.2.3 print_parser_errcode()
4.34 SRC/shell/shell.c File Reference
4.34.1 Macro Definition Documentation
4.34.1.1 MAX_BUFFER_SIZE
4.34.1.2 MAX_LINE_BUFFER_SIZE
4.34.1.3 PROMPT
4.34.2 Function Documentation
4.34.2.1 execute_command()
4.34.2.2 fill_buffer_until_full_or_newline()

4.34.2.3 free_job_ptr()	 251
4.34.2.4 setup_terminal_signal_handlers()	 252
4.34.2.5 shell()	 252
4.34.2.6 shell_sigint_handler()	 253
4.34.2.7 shell_sigstp_handler()	 254
4.34.2.8 u_execute_command()	 254
4.34.2.9 u_read_and_execute_script()	 255
4.34.3 Variable Documentation	 256
4.34.3.1 current_fg_pid	 256
4.34.3.2 input_fd_script	 256
4.34.3.3 job_list	 256
4.34.3.4 next_job_id	 256
4.34.3.5 output_fd_script	 257
4.34.3.6 script_fd	 257
4.35 SRC/shell/shell.h File Reference	 257
4.35.1 Function Documentation	 257
4.35.1.1 shell()	 257
4.36 SRC/shell/shell_built_ins.c File Reference	 259
4.36.1 Function Documentation	 261
4.36.1.1 orphan_child()	 261
4.36.1.2 u_bg()	 261
4.36.1.3 u_busy()	 261
4.36.1.4 u_cat()	 262
4.36.1.5 u_chmod()	 262
4.36.1.6 u_cp()	 263
4.36.1.7 u_echo()	 263
4.36.1.8 u_fg()	 263
4.36.1.9 u_jobs()	 264
4.36.1.10 u_kill()	 264
4.36.1.11 u_logout()	 265
4.36.1.12 u_ls()	 265
4.36.1.13 u_man()	 265
4.36.1.14 u_mv()	 266
4.36.1.15 u_nice()	 266
4.36.1.16 u_nice_pid()	 267
4.36.1.17 u_orphanify()	 267
4.36.1.18 u_ps()	 268
4.36.1.19 u_rm()	 268
4.36.1.20 u_sleep()	 268
4.36.1.21 u_touch()	 269
4.36.1.22 u_zombify()	 269
4.36.1.23 zombie_child()	 269

281

4.36.2 Variable Documentation	 269
4.36.2.1 get_associated_ufunc	 269
4.37 SRC/shell/shell_built_ins.h File Reference	 270
4.37.1 Function Documentation	 271
4.37.1.1 u_bg()	 272
4.37.1.2 u_busy()	 272
4.37.1.3 u_cat()	 272
4.37.1.4 u_chmod()	 273
4.37.1.5 u_cp()	 273
4.37.1.6 u_echo()	 274
4.37.1.7 u_fg()	 274
4.37.1.8 u_jobs()	 274
4.37.1.9 u_kill()	 275
4.37.1.10 u_logout()	 275
4.37.1.11 u_ls()	 276
4.37.1.12 u_man()	 276
4.37.1.13 u_mv()	 277
4.37.1.14 u_nice()	 277
4.37.1.15 u_nice_pid()	 278
4.37.1.16 u_orphanify()	 278
4.37.1.17 u_ps()	 278
4.37.1.18 u_rm()	 279
4.37.1.19 u_sleep()	 279
4.37.1.20 u_touch()	 280
4.37.1.21 u_zombify()	 280

Index

Chapter 1

Class Index

1.1 Class List

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

dir_entry_t	
Directory entry structure for files in the filesystem	5
fd_entry_t	
File descriptor entry structure for open files	7
job_st	8
parsed_command	10
pcb_st	
The PCB structure, which contains all the information about a process. Notably, it contains the	
thread handle, pid, parent pid, child pcbs, priority level, process state, command string, signals	
to be sent, input and output file descriptors, process status, sleeping status, and time to wake .	12
spthread_fwd_args_st	15
spthread_meta_st	17
spthread_signal_args_st	18
spthread_st	19
vec_st	20

2 Class Index

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

SRC/pennfat.c
SRC/pennos.c
SRC/fs/fat_routines.c
SRC/fs/fat_routines.h
SRC/fs/fs_helpers.c
SRC/fs/fs_helpers.h
SRC/fs/fs_kfuncs.c
SRC/fs/fs_kfuncs.h
SRC/fs/fs_syscalls.c
SRC/fs/fs_syscalls.h
SRC/kernel/kern_pcb.c
SRC/kernel/kern_pcb.h
SRC/kernel/kern_sys_calls.c
SRC/kernel/kern_sys_calls.h
SRC/kernel/logger.c
SRC/kernel/logger.h
SRC/kernel/scheduler.c
SRC/kernel/scheduler.h
SRC/kernel/signal.c
SRC/kernel/signal.h
SRC/kernel/stress.c
SRC/kernel/stress.h
SRC/lib/pennos-errno.c
SRC/lib/pennos-errno.h
SRC/lib/spthread.c
SRC/lib/spthread.h
SRC/lib/Vec.c
SRC/lib/Vec.h
SRC/shell/builtins.c
SRC/shell/builtins.h
SRC/shell/Job.h
SRC/shell/parser.c
SRC/shell/parser.h
SRC/shell/shell.c
SRC/shell/shell.h
SRC/shell/shell_built_ins.c
SRC/shell/shell_built_ins.h

File Index

Chapter 3

Class Documentation

3.1 dir_entry_t Struct Reference

Directory entry structure for files in the filesystem.

#include <fat_routines.h>

Public Attributes

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

3.1.1 Detailed Description

Directory entry structure for files in the filesystem.

Definition at line 46 of file fat_routines.h.

3.1.2 Member Data Documentation

3.1.2.1 firstBlock

uint16_t dir_entry_t::firstBlock

Definition at line 49 of file fat_routines.h.

3.1.2.2 mtime

```
time_t dir_entry_t::mtime
```

Definition at line 52 of file fat_routines.h.

3.1.2.3 name

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

Definition at line 47 of file fat_routines.h.

3.1.2.4 perm

```
uint8_t dir_entry_t::perm
```

Definition at line 51 of file fat_routines.h.

3.1.2.5 reserved

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

Definition at line 53 of file fat_routines.h.

3.1.2.6 size

```
uint32_t dir_entry_t::size
```

Definition at line 48 of file fat_routines.h.

3.1.2.7 type

```
uint8_t dir_entry_t::type
```

Definition at line 50 of file fat_routines.h.

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

• SRC/fs/fat_routines.h

3.2 fd_entry_t Struct Reference

File descriptor entry structure for open files.

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

Public Attributes

- int in_use
- int ref_count
- char filename [32]
- uint32_t size
- uint16_t first_block
- uint32_t position
- uint8 t mode

3.2.1 Detailed Description

File descriptor entry structure for open files.

Definition at line 59 of file fat_routines.h.

3.2.2 Member Data Documentation

3.2.2.1 filename

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

Definition at line 62 of file fat_routines.h.

3.2.2.2 first_block

```
uint16_t fd_entry_t::first_block
```

Definition at line 64 of file fat_routines.h.

3.2.2.3 in_use

```
int fd_entry_t::in_use
```

Definition at line 60 of file fat_routines.h.

3.2.2.4 mode

```
uint8_t fd_entry_t::mode
```

Definition at line 66 of file fat_routines.h.

3.2.2.5 position

```
uint32_t fd_entry_t::position
```

Definition at line 65 of file fat_routines.h.

3.2.2.6 ref_count

```
int fd_entry_t::ref_count
```

Definition at line 61 of file fat_routines.h.

3.2.2.7 size

```
uint32_t fd_entry_t::size
```

Definition at line 63 of file fat_routines.h.

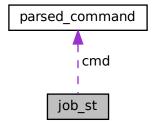
The documentation for this struct was generated from the following file:

• SRC/fs/fat_routines.h

3.3 job_st Struct Reference

```
#include <Job.h>
```

Collaboration diagram for job_st:



Public Attributes

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

3.3.1 Detailed Description

Definition at line 16 of file Job.h.

3.3.2 Member Data Documentation

3.3.2.1 cmd

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

Definition at line 18 of file Job.h.

3.3.2.2 finished_count

```
size_t job_st::finished_count
```

Definition at line 23 of file Job.h.

3.3.2.3 id

```
jid_t job_st::id
```

Definition at line 17 of file Job.h.

3.3.2.4 num_pids

```
size_t job_st::num_pids
```

Definition at line 21 of file Job.h.

3.3.2.5 pgid

```
pid_t job_st::pgid
```

Definition at line 22 of file Job.h.

3.3.2.6 pids

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

Definition at line 19 of file Job.h.

3.3.2.7 state

```
job_state_t job_st::state
```

Definition at line 20 of file Job.h.

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

SRC/shell/Job.h

3.4 parsed_command Struct Reference

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

Public Attributes

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

3.4.1 Detailed Description

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

Definition at line 36 of file parser.h.

3.4.2 Member Data Documentation

3.4.2.1 commands

char** parsed_command::commands[]

Definition at line 56 of file parser.h.

3.4.2.2 is_background

bool parsed_command::is_background

Definition at line 39 of file parser.h.

3.4.2.3 is_file_append

bool parsed_command::is_file_append

Definition at line 43 of file parser.h.

3.4.2.4 num_commands

size_t parsed_command::num_commands

Definition at line 52 of file parser.h.

3.4.2.5 stdin_file

const char* parsed_command::stdin_file

Definition at line 46 of file parser.h.

3.4.2.6 stdout_file

const char* parsed_command::stdout_file

Definition at line 49 of file parser.h.

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

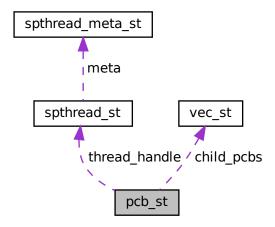
· SRC/shell/parser.h

3.5 pcb_st Struct Reference

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

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

Collaboration diagram for pcb_st:



Public Attributes

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

3.5.1 Detailed Description

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

Definition at line 30 of file kern_pcb.h.

3.5.2 Member Data Documentation

3.5.2.1 child_pcbs

```
Vec pcb_st::child_pcbs
```

Definition at line 36 of file kern_pcb.h.

3.5.2.2 cmd_str

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

Definition at line 42 of file kern_pcb.h.

3.5.2.3 fd_table

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

Definition at line 61 of file kern_pcb.h.

3.5.2.4 input_fd

```
int pcb_st::input_fd
```

Definition at line 48 of file kern_pcb.h.

3.5.2.5 is_sleeping

```
bool pcb_st::is_sleeping
```

Definition at line 58 of file kern_pcb.h.

3.5.2.6 output_fd

```
int pcb_st::output_fd
```

Definition at line 49 of file kern_pcb.h.

3.5.2.7 par_pid

```
pid_t pcb_st::par_pid
```

Definition at line 34 of file kern_pcb.h.

3.5.2.8 pid

```
pid_t pcb_st::pid
```

Definition at line 33 of file kern_pcb.h.

3.5.2.9 priority

```
int pcb_st::priority
```

Definition at line 38 of file kern_pcb.h.

3.5.2.10 process_state

char pcb_st::process_state

Definition at line 39 of file kern_pcb.h.

3.5.2.11 process_status

```
int pcb_st::process_status
```

Definition at line 51 of file kern_pcb.h.

3.5.2.12 signals

```
bool pcb_st::signals[3]
```

Definition at line 44 of file kern_pcb.h.

3.5.2.13 thread_handle

```
spthread_t pcb_st::thread_handle
```

Definition at line 31 of file kern_pcb.h.

3.5.2.14 time_to_wake

```
int pcb_st::time_to_wake
```

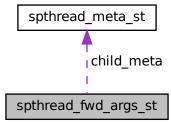
Definition at line 59 of file kern_pcb.h.

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

• SRC/kernel/kern_pcb.h

3.6 spthread_fwd_args_st Struct Reference

Collaboration diagram for spthread_fwd_args_st:



Public Attributes

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

3.6.1 Detailed Description

Definition at line 22 of file spthread.c.

3.6.2 Member Data Documentation

3.6.2.1 actual_arg

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

Definition at line 27 of file spthread.c.

3.6.2.2 actual_routine

```
pthread_fn spthread_fwd_args_st::actual_routine
```

Definition at line 26 of file spthread.c.

3.6.2.3 child_meta

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

Definition at line 40 of file spthread.c.

3.6.2.4 setup_cond

 $\verb|pthread_cond_t| spthread_fwd_args_st::setup_cond|$

Definition at line 37 of file spthread.c.

3.6.2.5 setup_done

bool spthread_fwd_args_st::setup_done

Definition at line 35 of file spthread.c.

3.6.2.6 setup_mutex

```
pthread_mutex_t spthread_fwd_args_st::setup_mutex
```

Definition at line 36 of file spthread.c.

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

• SRC/lib/spthread.c

3.7 spthread_meta_st Struct Reference

Public Attributes

- sigset_t suspend_set
- volatile sig_atomic_t state
- pthread_mutex_t meta_mutex

3.7.1 Detailed Description

Definition at line 57 of file spthread.c.

3.7.2 Member Data Documentation

3.7.2.1 meta_mutex

 $\verb|pthread_mutex_t spthread_meta_st::meta_mutex|\\$

Definition at line 71 of file spthread.c.

3.7.2.2 state

volatile sig_atomic_t spthread_meta_st::state

Definition at line 68 of file spthread.c.

3.7.2.3 suspend_set

```
sigset_t spthread_meta_st::suspend_set
```

Definition at line 59 of file spthread.c.

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

• SRC/lib/spthread.c

3.8 spthread_signal_args_st Struct Reference

Public Attributes

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

3.8.1 Detailed Description

Definition at line 46 of file spthread.c.

3.8.2 Member Data Documentation

3.8.2.1 ack

 $\verb|volatile| sig_atomic_t| spthread_signal_args_st::ack|$

Definition at line 48 of file spthread.c.

3.8.2.2 shutup_mutex

```
\verb|pthread_mutex_t spthread_signal_args_st:: shutup_mutex|
```

Definition at line 49 of file spthread.c.

3.8.2.3 signal

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

Definition at line 47 of file spthread.c.

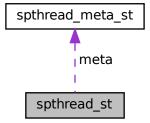
The documentation for this struct was generated from the following file:

• SRC/lib/spthread.c

3.9 spthread_st Struct Reference

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

Collaboration diagram for spthread_st:



Public Attributes

- pthread_t thread
- spthread_meta_t * meta

3.9.1 Detailed Description

Definition at line 28 of file spthread.h.

20 Class Documentation

3.9.2 Member Data Documentation

3.9.2.1 meta

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

Definition at line 30 of file spthread.h.

3.9.2.2 thread

```
pthread_t spthread_st::thread
```

Definition at line 29 of file spthread.h.

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

• SRC/lib/spthread.h

3.10 vec_st Struct Reference

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

Public Attributes

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

3.10.1 Detailed Description

Definition at line 10 of file Vec.h.

3.10.2 Member Data Documentation

3.10.2.1 capacity

```
size_t vec_st::capacity
```

Definition at line 13 of file Vec.h.

3.10.2.2 data

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

Definition at line 11 of file Vec.h.

3.10.2.3 ele_dtor_fn

```
ptr_dtor_fn vec_st::ele_dtor_fn
```

Definition at line 14 of file Vec.h.

3.10.2.4 length

```
size_t vec_st::length
```

Definition at line 12 of file Vec.h.

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

• SRC/lib/Vec.h

22 Class Documentation

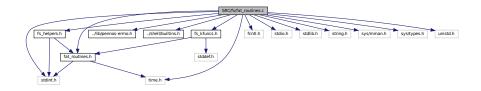
Chapter 4

File Documentation

4.1 SRC/fs/fat_routines.c File Reference

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

Include dependency graph for fat_routines.c:



Functions

- int mkfs (const char *fs_name, int num_blocks, int blk_size)
 - Creates a PennFAT filesystem in the file named fs_name at the OS-level.
- int mount (const char *fs_name)

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

• int unmount ()

Unmounts the current filesystem and reset variables.

void * cat (void *arg)

Concatenates and displays files.

```
    void * Is (void *arg)
        Searches root directory and lists all files in the directory.
    void * touch (void *arg)
        Creates files or updates timestamps.
    void * mv (void *arg)
        Renames files.
    void * cp (void *arg)
        Copies the source file to the destination.
    void * rm (void *arg)
        Removes files.
    void * chmod (void *arg)
        Changes the permissions of a file.
    void * cmpctdir (void *arg)
```

Implements compaction of root directory.

4.1.1 Function Documentation

4.1.1.1 cat()

Concatenates and displays files.

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

Usage formats:

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

Parameters

```
arg Arguments array (command line arguments)
```

Returns

void pointer (unused)

Definition at line 209 of file fat_routines.c.

```
209
210
      char** args = (char**)arg;
211
212
       \ensuremath{//} verify that the file system is mounted
      if (!is_mounted) {
  P_ERRNO = P_EFS_NOT_MOUNTED;
213
214
        u_perror("cat");
215
216
         return NULL;
217
218
219
      \ensuremath{//} early return if there is nothing after cat
      if (args[1] == NULL) {
  P_ERRNO = P_EINVAL;
  u_perror("cat");
220
221
222
223
         return NULL;
224
225
226
      // check for output file with -w or -a flag
      int out_fd = -1;
227
228
      int out_mode = 0;
229
230
      \ensuremath{//} scan arguments and determine output fd and output mode
      int i;
for (i = 1; args[i] != NULL; i++) {
   if (strcmp(args[i], "-w") == 0 && args[i + 1] != NULL) {
2.31
2.32
233
234
          out_mode = F_WRITE;
235
           out_fd = k_open(args[i + 1], F_WRITE);
          if (out_fd < 0) {
   u_perror("cat");</pre>
236
237
            return NULL;
238
239
240
           break;
241
        } else if (strcmp(args[i], "-a") == 0 && args[i + 1] != NULL) {
          out_mode = F_APPEND;
out_fd = k_open(args[i + 1], F_APPEND);
if (out_fd < 0) {
   u_perror("cat");
   return NULL;</pre>
242
243
244
245
246
247
248
           break;
249
        }
      }
2.50
251
252
      // if no output redirection found, use STDOUT
      if (out_fd < 0) {</pre>
253
        out_fd = STDOUT_FILENO;
254
255
256
      257
258
259
260
        char buffer[1024];
261
2.62
        while (1) {
          ssize_t bytes_read = k_read(STDIN_FILENO, buffer, sizeof(buffer));
263
264
265
           if (bytes_read < 0) {</pre>
266
            u_perror("cat");
                (out_fd != STDOUT_FILENO) {
267
268
               k_close(out_fd);
             }
269
270
             return NULL;
271
           }
272
273
           if (bytes_read == 0) {
274
            break;
275
          }
276
277
           if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
278
            u_perror("cat");
             if (out_fd != STDOUT_FILENO) {
279
280
               k_close(out_fd);
281
             return NULL;
282
283
          }
284
285
286
        if (out_fd != STDOUT_FILENO) {
287
288
289
         return NULL;
290
291
292
      // handle concatenating one or more files: cat FILE ... [-w/-a OUTPUT_FILE]
      int start = 1;
int end = i - 1;
293
2.94
295
```

```
296
      if (out_mode != 0) {
297
        end = i - 1; // skip the output redirection arguments
298
299
      // process each input file
for (i = start; i <= end; i++) {
    // skip the redirection flags and their arguments</pre>
300
301
302
303
         if (strcmp(args[i], "-w") == 0 || strcmp(args[i], "-a") == 0) {
         i++;
304
305
           continue;
306
307
308
         // open the current input file
309
         int in_fd = k_open(args[i], F_READ);
310
         if (in_fd < 0) {
         u_perror("cat");
311
312
           continue;
313
314
315
         // use lseek to get the size of in_fd
         off_t in_fd_size = k_lseek(in_fd, 0, SEEK_END);
if (in_fd_size == -1) {
316
317
          k_close(in_fd);
u_perror("cat");
318
319
320
           continue;
321
322
323
         // use lseek to reset position to 0 for reading
         if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
324
325
          k close(in_fd);
           u_perror("cat");
326
327
           continue;
328
329
330
         \ensuremath{//} copy file content to output
         char* buffer = (char*)malloc(block_size);
331
         if (buffer == NULL) {
   P_ERRNO = P_EMALLOC;
332
333
334
           k_close(in_fd);
335
           u_perror("cat");
336
           continue;
337
338
339
         int bytes_read;
340
         ssize_t bytes_remaining = in_fd_size;
341
         while (bytes_remaining > 0) {
   ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
342
343
344
           bytes_read = k_read(in_fd, buffer, bytes_to_read);
345
346
           if (bytes_read <= 0) {</pre>
347
348
349
350
           if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
351
            free (buffer);
352
             k_close(in_fd);
353
             u_perror("cat");
354
355
356
           bytes_remaining -= bytes_read;
357
358
359
360
         // read error
361
         if (bytes_read < 0) {</pre>
362
          free (buffer);
363
           k close(in fd);
364
          u_perror("cat");
365
           continue;
366
367
368
        k_close(in_fd);
369
        free (buffer);
370
371
372
       // close output file if not stdout
373
      if (out_fd != STDOUT_FILENO) {
374
        k_close(out_fd);
      }
375
376
      return NULL;
378 }
```

4.1.1.2 chmod()

Changes the permissions of a file.

Changes file permissions.

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

Definition at line 751 of file fat_routines.c.

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

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

4.1.1.3 cmpctdir()

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

Implements compaction of root directory.

Compacts the root directory by removing all deleted entries.

Definition at line 832 of file fat_routines.c.

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

4.1.1.4 cp()

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

Copies the source file to the destination.

Copies files.

Definition at line 621 of file fat_routines.c.

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

```
640
         u_perror("cp");
641
         return NULL;
642
643
       return NULL;
644
645
     // cp SOURCE -h DEST
646
647
     if (args[2] != NULL && strcmp(args[2], "-h") == 0) {
     if (args[3] == NULL) {
  P_ERRNO = P_EINVAL;
648
649
        u_perror("cp");
650
651
         return NULL;
652
653
654
       if (copy_pennfat_to_host(args[1], args[3]) != 0) {
       u_perror("cp");
655
656
        return NULL;
       }
657
658
       return NULL;
659
     }
660
661
     // cp SOURCE DEST
     662
663
664
665
        u_perror("cp");
666
         return NULL;
667
668
       return NULL;
669
     }
670
     P_ERRNO = P_EUNKNOWN;
u_perror("cp");
671
672
673
     return NULL;
674 }
```

4.1.1.5 ls()

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

Searches root directory and lists all files in the directory.

Lists files in the current directory.

Definition at line 383 of file fat_routines.c.

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

```
413
           // check if we've reached the end of directory
414
           if (dir_entry.name[0] == 0) {
415
             break;
           }
416
417
           // skip deleted entries
418
           if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
419
420
            offset += sizeof(dir_entry);
421
             continue;
422
423
           // format permission string
424
           char perm_str[4] =
425
          if (dir_entry.perm & PERM_READ)
  perm_str[0] = 'r';
426
427
          if (dir_entry.perm & PERM_WRITE)
  perm_str[1] = 'w';
if (dir_entry.perm & PERM_EXEC)
  perm_str[2] = 'x';
428
429
430
431
432
433
           // format time
           struct tm* tm_info = localtime(&dir_entry.mtime);
434
435
           char time_str[50];
           strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);
436
437
438
           // print entry details
439
           char buffer[128];
440
           int len;
441
           if (dir_entry.firstBlock == 0) {
            len = snprintf(buffer, sizeof(buffer), " -%s- %6d %s %s\n",
442
443
                      perm_str, dir_entry.size, time_str, dir_entry.name);
444
445
            len = snprintf(buffer, sizeof(buffer), "%2d -%s- %6d %s %s\n",
446
                      dir_entry.firstBlock, perm_str, dir_entry.size, time_str, dir_entry.name);
447
448
           if (len < 0 || len >= (int)sizeof(buffer)) {
   P_ERRNO = P_EUNKNOWN;
449
450
451
             u_perror("ls");
452
             return NULL;
453
454
           if (k_write(STDOUT_FILENO, buffer, len) != len) {
455
456
            P_ERRNO = P_EWRITE;
             u_perror("ls");
457
458
             return NULL;
459
460
461
          offset += sizeof(dir_entry);
462
463
        // move to the next block if there is one
if (fat[current_block] != FAT_EOF) {
464
465
466
         current_block = fat[current_block];
467
           continue;
468
469
470
         // no more blocks to search
471
472
473
474
      return NULL;
```

4.1.1.6 mkfs()

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

Creates a PennFAT filesystem in the file named fs_name.

Definition at line 29 of file fat_routines.c.

```
{
     // validate arguments
30
     if (num_blocks < 1 || num_blocks > 32) {
   P_ERRNO = P_EINVAL;
31
32
       return -1;
33
34
35
     if (blk_size < 0 || blk_size > 4) {
36
       P_ERRNO = P_EINVAL;
37
        return -1;
38
39
     // determine the file system size
40
     int block_sizes[] = {256, 512, 1024, 2048, 4096};
41
     int actual_block_size = block_sizes[blk_size];
43
     int fat_size = num_blocks * actual_block_size;
44
     int fat_entries = fat_size / 2;
45
     int num data blocks =
          (num blocks == 32)
46
              ? fat_entries - 2
               : fat_entries - 1; // note: first entry is reserved for metadata!
49
     size_t filesystem_size = fat_size + (actual_block_size * num_data_blocks);
50
     \ensuremath{//} create the file for the filesystem
51
     int fd = open(fs_name, O_RDWR | O_CREAT | O_TRUNC, 0644);
if (fd == -1) {
   P_ERRNO = P_EOPEN;
52
53
55
       return -1;
     }
56
57
     // extend the file to the required size
if (ftruncate(fd, filesystem_size) == -1) {
58
59
60
       P_ERRNO = P_EFUNC;
        close(fd);
62
        return -1;
     }
63
64
     // allocate the FAT
65
     uint16_t* temp_fat = (uint16_t*)calloc(fat_entries, sizeof(uint16_t));
66
     if (!temp_fat) {
68
      P_ERRNO = P_EMALLOC;
69
       close(fd);
70
       return -1;
71
     // initialize FAT entries to their correct values
74
     temp_fat[0] = (num_blocks « 8) | blk_size;
     temp_fat[1] = FAT_EOF;
for (int i = 2; i < fat_entries; i++) {
  temp_fat[i] = FAT_FREE;</pre>
75
76
77
78
80
     // write the FAT to the file
81
     if (write(fd, temp_fat, fat_size) != fat_size) {
      P_ERRNO = P_EWRITE;
82
83
       free(temp_fat);
84
        close(fd);
       return -1;
86
87
     \ensuremath{//} initialize the root directory + write to memory
88
     uint8_t* root_dir = (uint8_t*)calloc(actual_block_size, 1);
if (lseek(fd, fat_size, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
89
90
       free(temp_fat);
93
        free (root_dir);
94
       close(fd);
95
       return -1;
96
97
     if (write(fd, root_dir, actual_block_size) != actual_block_size) {
      P_ERRNO = P_EWRITE;
99
        free(temp_fat);
100
        free(root_dir);
101
        close(fd);
102
         return -1;
103
104
105
      // clean up
106
      free(temp_fat);
107
      free (root_dir);
108
     close(fd):
109
      return 0;
110 }
```

4.1.1.7 mount()

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

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

Definition at line 115 of file fat_routines.c.

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

4.1.1.8 mv()

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

Renames files.

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

Usage: mv SOURCE DEST

Parameters

arg | Arguments array (command line arguments)

Returns

void pointer (unused)

```
Definition at line 543 of file fat_routines.c.
```

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

```
614
615 return NULL;
616 }
```

4.1.1.9 rm()

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

Removes files.

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

Parameters

arg | Arguments array (command line arguments)

Returns

679

void pointer (unused)

Definition at line 679 of file fat_routines.c.

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

```
char deleted = 1;
                                  // mark as deleted
726
          if (write(fs_fd, &deleted, sizeof(deleted)) != sizeof(deleted)) {
727
            P_ERRNO = P_EWRITE;
           u_perror("rm");
728
729
            continue;
730
731
732
         // free the FAT chain for this file
         wint16_t block = entry.firstBlock;
while (block != FAT_FREE && block != FAT_EOF) {
   uint16_t next_block = fat[block];
   fat[block] = FAT_FREE;
733
734
735
736
737
            block = next_block;
738
739
740
       return NULL:
741
742 }
```

4.1.1.10 touch()

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

Creates files or updates timestamps.

Creates empty files or updates timestamps.

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

Definition at line 483 of file fat routines.c.

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

```
}
528
           \ensuremath{//} add the file entry to root directory
529
          if (add_file_entry(args[i], 0, 0, TYPE_REGULAR, PERM_READ_WRITE) == -1) {
  u_perror("touch");
530
531
532
             continue:
533
534
535 }
536
      return NULL;
537
538 }
```

4.1.1.11 unmount()

```
int unmount ( )
```

Unmounts the current filesystem and reset variables.

Unmounts the currently mounted filesystem.

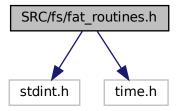
Definition at line 169 of file fat routines.c.

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

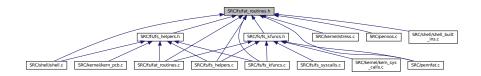
4.2 SRC/fs/fat_routines.h File Reference

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

Include dependency graph for fat_routines.h:



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



Classes

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

File descriptor entry structure for open files.

Macros

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

Functions

```
• int mkfs (const char *fs_name, int num_blocks, int block_size)
```

Creates a PennFAT filesystem in the file named fs name.

int mount (const char *fs_name)

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

• int unmount ()

Unmounts the currently mounted filesystem.

void * cat (void *arg)

Concatenates and displays files.

void * Is (void *arg)

Lists files in the current directory.

void * touch (void *arg)

Creates empty files or updates timestamps.

void * mv (void *arg)

Renames files.

void * cp (void *arg)

Copies files.

void * rm (void *arg)

Removes files.

void * chmod (void *arg)

Changes file permissions.

void * cmpctdir (void *arg)

Compacts the root directory by removing all deleted entries.

4.2.1 Macro Definition Documentation

4.2.1.1 F APPEND

#define F_APPEND 0x04

Definition at line 37 of file fat routines.h.

4.2.1.2 F READ

#define F_READ 0x01

Definition at line 35 of file fat_routines.h.

4.2.1.3 F_WRITE

#define F_WRITE 0x02

Definition at line 36 of file fat_routines.h.

4.2.1.4 FAT_EOF

#define FAT_EOF 0xFFFF

Definition at line 16 of file fat_routines.h.

4.2.1.5 **FAT_FREE**

#define FAT_FREE 0x0000

Definition at line 17 of file fat_routines.h.

4.2.1.6 PERM_EXEC

#define PERM_EXEC 4

Definition at line 29 of file fat_routines.h.

4.2.1.7 PERM NONE

#define PERM_NONE 0

Definition at line 26 of file fat_routines.h.

4.2.1.8 PERM_READ

#define PERM_READ 2

Definition at line 28 of file fat_routines.h.

4.2.1.9 PERM_READ_EXEC

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

Definition at line 31 of file fat_routines.h.

4.2.1.10 PERM_READ_WRITE

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

Definition at line 30 of file fat_routines.h.

4.2.1.11 PERM_READ_WRITE_EXEC

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

Definition at line 32 of file fat_routines.h.

4.2.1.12 **PERM_WRITE**

```
#define PERM_WRITE 1
```

Definition at line 27 of file fat_routines.h.

4.2.1.13 TYPE DIRECTORY

```
#define TYPE_DIRECTORY 2
```

Definition at line 22 of file fat_routines.h.

4.2.1.14 TYPE_REGULAR

```
#define TYPE_REGULAR 1
```

Definition at line 21 of file fat_routines.h.

4.2.1.15 TYPE_SYMLINK

```
#define TYPE_SYMLINK 4
```

Definition at line 23 of file fat_routines.h.

4.2.1.16 TYPE_UNKNOWN

```
#define TYPE_UNKNOWN 0
```

Definition at line 20 of file fat_routines.h.

4.2.2 Function Documentation

4.2.2.1 cat()

Concatenates and displays files.

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

Usage formats:

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

Parameters

arg Arguments array (command line arguments)

Returns

void pointer (unused)

Definition at line 209 of file fat_routines.c.

```
209
210
      char** args = (char**)arg;
211
212
      \ensuremath{//} verify that the file system is mounted
      if (!is_mounted) {
   P_ERRNO = P_EFS_NOT_MOUNTED;
213
214
        u_perror("cat");
215
216
        return NULL;
217
218
      // early return if there is nothing after cat
219
      if (args[1] == NULL) {
  P_ERRNO = P_EINVAL;
  u_perror("cat");
220
221
222
223
        return NULL;
224
225
226
      // check for output file with -w or -a flag
      int out_fd = -1;
227
228
      int out_mode = 0;
229
230
      \ensuremath{//} scan arguments and determine output fd and output mode
      int i;
for (i = 1; args[i] != NULL; i++) {
   if (strcmp(args[i], "-w") == 0 && args[i + 1] != NULL) {
2.31
232
233
234
          out_mode = F_WRITE;
235
           out_fd = k_open(args[i + 1], F_WRITE);
          if (out_fd < 0) {
   u_perror("cat");</pre>
236
237
            return NULL;
238
239
240
          break;
241
        } else if (strcmp(args[i], "-a") == 0 && args[i + 1] != NULL) {
          out_mode = F_APPEND;
out_fd = k_open(args[i + 1], F_APPEND);
if (out_fd < 0) {</pre>
242
243
244
           u_perror("cat");
return NULL;
245
246
247
248
          break;
249
        }
      }
2.50
251
252
      // if no output redirection found, use STDOUT
      if (out_fd < 0) {</pre>
253
        out_fd = STDOUT_FILENO;
254
255
256
      257
258
259
260
        char buffer[1024];
261
2.62
        while (1) {
          ssize_t bytes_read = k_read(STDIN_FILENO, buffer, sizeof(buffer));
263
264
265
          if (bytes_read < 0) {</pre>
266
            u_perror("cat");
                (out_fd != STDOUT_FILENO) {
267
268
               k_close(out_fd);
             }
269
270
             return NULL;
271
272
273
           if (bytes_read == 0) {
274
            break;
275
          }
276
277
           if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
            u_perror("cat");
if (out_fd != STDOUT_FILENO) {
278
279
280
               k_close(out_fd);
281
             return NULL;
282
283
          }
284
285
286
        if (out_fd != STDOUT_FILENO) {
287
288
289
        return NULL;
290
291
292
      // handle concatenating one or more files: cat FILE ... [-w/-a OUTPUT_FILE]
293
      int start = 1;
      int end = i - 1;
2.94
295
```

```
296
      if (out_mode != 0) {
297
        end = i - 1; // skip the output redirection arguments
298
299
      // process each input file
for (i = start; i <= end; i++) {
    // skip the redirection flags and their arguments</pre>
300
301
302
303
         if (strcmp(args[i], "-w") == 0 || strcmp(args[i], "-a") == 0) {
         i++;
304
305
           continue;
306
307
308
         // open the current input file
309
         int in_fd = k_open(args[i], F_READ);
310
         if (in_fd < 0) {</pre>
         u_perror("cat");
311
312
           continue;
313
314
315
         // use lseek to get the size of in_fd
316
         off_t in_fd_size = k_lseek(in_fd, 0, SEEK_END);
         if (in_fd_size == -1) {
317
          k_close(in_fd);
u_perror("cat");
318
319
320
           continue;
321
322
323
         // use lseek to reset position to 0 for reading
         if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
324
325
          k close(in_fd);
           u_perror("cat");
326
327
           continue;
328
329
330
         \ensuremath{//} copy file content to output
         char* buffer = (char*)malloc(block_size);
331
         if (buffer == NULL) {
   P_ERRNO = P_EMALLOC;
332
333
334
           k_close(in_fd);
335
          u_perror("cat");
336
           continue;
337
338
339
         int bytes_read;
340
         ssize_t bytes_remaining = in_fd_size;
341
         while (bytes_remaining > 0) {
   ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
342
343
344
           bytes_read = k_read(in_fd, buffer, bytes_to_read);
345
346
           if (bytes_read <= 0) {</pre>
347
348
349
350
           if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
351
            free (buffer);
352
             k_close(in_fd);
353
             u_perror("cat");
354
355
356
          bytes_remaining -= bytes_read;
357
358
359
360
         // read error
361
         if (bytes_read < 0) {</pre>
362
          free (buffer);
363
           k close(in fd);
364
          u_perror("cat");
365
           continue;
366
367
368
        k_close(in_fd);
369
        free (buffer);
370
371
372
      // close output file if not stdout
373
      if (out_fd != STDOUT_FILENO) {
374
        k_close(out_fd);
      }
375
376
      return NULL;
378 }
```

4.2.2.2 chmod()

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

Changes file permissions.

Modifies the permissions of the specified file.

Usage formats:

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

Parameters

```
arg Arguments array (command line arguments)
```

Returns

void pointer (unused)

Changes file permissions.

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

Definition at line 751 of file fat_routines.c.

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

```
} else {
782
             new_perm &= ~PERM_READ;
783
784
            break;
          case 'w':
785
          if (perm_str[0] == '+') {
786
             new_perm |= PERM_WRITE;
788
789
             new_perm &= ~PERM_WRITE;
790
791
           break:
        case 'x':
792
          if (perm_str[0] == '+') {
793
794
             new_perm |= PERM_EXEC;
795
796
797
             new_perm &= ~PERM_EXEC;
798
            break;
799
          default:
800
           P_ERRNO = P_EINVAL;
801
            return NULL;
802
       i++;
803
804
      }
805
806
      // Update the directory entry
807
      dir_entry.perm = new_perm;
808
      dir_entry.mtime = time(NULL);
809
810
      // Seek to the entry's position
      if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
811
      P_ERRNO = P_ELSEEK;
return NULL;
812
813
814
815
      // Write the updated entry back
816
      if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
    P_ERRNO = P_EWRITE;
817
818
819
        return NULL;
820
821
     return NULL;
822
823 }
```

4.2.2.3 cmpctdir()

Compacts the root directory by removing all deleted entries.

Parameters

arg Arguments array (command line arguments)

Returns

void pointer (unused)

Compacts the root directory by removing all deleted entries.

Definition at line 832 of file fat_routines.c.

```
839    if (compact_directory() != 0) {
840        u_perror("cmpctdir");
841    }
842
843    return NULL;
844 }
```

4.2.2.4 cp()

Copies files.

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

Usage formats:

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

Parameters

arg | Arguments array (command line arguments)

Returns

return 0 on success, -1 on error

Copies files.

Definition at line 621 of file fat routines.c.

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

```
648
      if (args[3] == NULL) {
       P_ERRNO = P_EINVAL;
u_perror("cp");
649
650
        return NULL;
651
652
653
654
      if (copy_pennfat_to_host(args[1], args[3]) != 0) {
       u_perror("cp");
return NULL;
655
656
657
      return NULL;
658
659
660
     661
662
663
664
665
666
        return NULL;
667
668
      return NULL;
669
670
671
    P_ERRNO = P_EUNKNOWN;
672
    u_perror("cp");
    return NULL;
674 }
```

4.2.2.5 ls()

Lists files in the current directory.

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

Parameters

arg | Arguments array (command line arguments)

Returns

0 on success, -1 on error

Lists files in the current directory.

Definition at line 383 of file fat_routines.c.

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

```
401
        }
402
403
        offset = 0;
404
405
        // search current block
406
        while (offset < block size) {
407
          if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
             P_ERRNO = P_EREAD;
u_perror("ls");
408
409
410
            return NULL;
411
412
413
           // check if we've reached the end of directory
414
           if (dir_entry.name[0] == 0) {
415
            break;
416
417
           // skip deleted entries
418
           if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
419
            offset += sizeof(dir_entry);
420
421
422
423
          // format permission string
char perm_str[4] = "---";
424
425
          if (dir_entry.perm & PERM_READ)
  perm_str[0] = 'r';
426
427
          if (dir_entry.perm & PERM_WRITE)
  perm_str[1] = 'w';
if (dir_entry.perm & PERM_EXEC)
  perm_str[2] = 'x';
428
429
430
431
432
433
           // format time
434
           struct tm* tm_info = localtime(&dir_entry.mtime);
435
           char time_str[50];
           strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);
436
437
438
           // print entry details
439
           char buffer[128];
440
           int len;
441
           if (dir_entry.firstBlock == 0) {
            len = snprintf(buffer, sizeof(buffer), " -%s- %6d %s %s\n",
442
                     perm_str, dir_entry.size, time_str, dir_entry.name);
443
444
           } else {
445
            len = snprintf(buffer, sizeof(buffer), "%2d -%s- %6d %s %s\n",
446
                      dir_entry.firstBlock, perm_str, dir_entry.size, time_str, dir_entry.name);
447
448
           if (len < 0 || len >= (int)sizeof(buffer)) {
449
450
            P_ERRNO = P_EUNKNOWN;
             u_perror("ls");
451
452
            return NULL;
453
454
           if (k_write(STDOUT_FILENO, buffer, len) != len) {
455
            P_ERRNO = P_EWRITE;
u_perror("ls");
456
457
458
             return NULL;
459
460
          offset += sizeof(dir_entry);
461
462
463
464
         // move to the next block if there is one
465
         if (fat[current_block] != FAT_EOF)
466
         current_block = fat[current_block];
467
           continue;
468
469
470
         // no more blocks to search
471
472
473
474
      return NULL;
475 }
```

4.2.2.6 mkfs()

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

Creates a PennFAT filesystem in the file named fs_name.

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

Parameters

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

Creates a PennFAT filesystem in the file named fs_name.

Definition at line 29 of file fat routines.c.

```
30
     // validate arguments
31
     if (num_blocks < 1 || num_blocks > 32) {
      P_ERRNO = P_EINVAL;
return -1;
32
33
34
35
    if (blk_size < 0 || blk_size > 4) {
       P_ERRNO = P_EINVAL;
37
       return -1;
38
39
    // determine the file system size
int block_sizes[] = {256, 512, 1024, 2048, 4096};
40
    int actual_block_size = block_sizes[blk_size];
42
     int fat_size = num_blocks * actual_block_size;
44
    int fat_entries = fat_size / 2;
4.5
    int num_data_blocks =
          (num blocks == 32)
46
             ? fat entries - 2
47
              : fat_entries - 1; // note: first entry is reserved for metadata!
48
49
    size_t filesystem_size = fat_size + (actual_block_size * num_data_blocks);
50
51
     \ensuremath{//} create the file for the filesystem
    int fd = open(fs_name, O_RDWR | O_CREAT | O_TRUNC, 0644);
52
    if (fd == -1) {
53
     P_ERRNO = P_EOPEN;
54
       return -1;
57
    // extend the file to the required size
if (ftruncate(fd, filesystem_size) == -1) {
58
59
60
      P_ERRNO = P_EFUNC;
       close(fd);
61
       return -1;
    }
63
64
    // allocate the FAT
65
    uint16_t* temp_fat = (uint16_t*)calloc(fat_entries, sizeof(uint16_t));
66
    if (!temp_fat) {
68
       P_ERRNO = P_EMALLOC;
69
       close(fd);
70
       return -1;
71
72
73
    // initialize FAT entries to their correct values
     temp_fat[0] = (num_blocks « 8) | blk_size;
    temp_fat[1] = FAT_EOF;
for (int i = 2; i < fat_entries; i++) {
75
76
      temp_fat[i] = FAT_FREE;
77
78
    // write the FAT to the file
     if (write(fd, temp_fat, fat_size) != fat_size) {
82
       P_ERRNO = P_EWRITE;
8.3
       free(temp_fat);
       close(fd);
84
85
       return -1;
```

```
// initialize the root directory + write to memory
    uint8_t* root_dir = (uint8_t*)calloc(actual_block_size, 1);
if (lseek(fd, fat_size, SEEK_SET) == -1) {
89
90
91
        P ERRNO = P ELSEEK;
       free(temp_fat);
92
       free(root_dir);
93
94
       close(fd);
95
        return -1;
96
     if (write(fd, root_dir, actual_block_size) != actual_block_size) {
   P_ERRNO = P_EWRITE;
97
98
99
        free(temp_fat);
100
        free (root_dir);
101
         close(fd);
102
         return -1;
103
104
105
      // clean up
106
      free(temp_fat);
107
       free (root_dir);
108
      close(fd);
109
      return 0;
110 }
```

4.2.2.7 mount()

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

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

Parameters

fs_name The name of the filesystem file to mount.

Returns

0 on success, -1 on failure with P ERRNO set.

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

Definition at line 115 of file fat_routines.c.

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

```
136
        }
137
138
        \ensuremath{//} extract FAT region size information
        num_fat_blocks = (config » 8) & 0xFF; // MSB
139
       int block_size_config = config & 0xFF; // LSB
int block_sizes[] = {256, 512, 1024, 2048, 4096};
block_size = block_sizes[block_size_config];
140
141
142
143
        fat_size = num_fat_blocks * block_size;
144
        // map the FAT region into memory
if (lseek(fs_fd, 0, SEEK_SET) == -1) {
  P_ERRNO = P_ELSEEK;
145
146
147
148
          close(fs_fd);
149
          fs_fd = -1;
150
          return -1;
151
152
        fat = mmap(NULL, fat_size, PROT_READ | PROT_WRITE, MAP_SHARED, fs_fd, 0);
if (fat == MAP_FAILED) {
153
154
155
          P_ERRNO = P_EMAP;
156
          close(fs_fd);
          fs_fd = -1;
return -1;
157
158
159
160
       init_fd_table(fd_table); // initialize the file descriptor table
161
162
163 return 0;
164 }
```

4.2.2.8 mv()

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

Renames files.

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

Usage: mv SOURCE DEST

Parameters

arg | Arguments array (command line arguments)

Returns

void pointer (unused)

Definition at line 543 of file fat_routines.c.

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

```
560
      char* source = args[1];
561
      char* dest = args[2];
562
563
       // check if they're trying to rename to the same name
564
      if (strcmp(source, dest) == 0) {
565
        return NULL;
566
567
568
      // check if source file exists
569
      dir_entry_t source_entry;
      int source_offset = find_file(source, &source_entry);
if (source_offset < 0) {</pre>
570
571
        u_perror("mv");
572
        return NULL;
573
574
575
576
      // check if the destination file already exists
      dir_entry_t dest_entry;
int dest_offset = find_file(dest, &dest_entry);
577
579
580
       // destination file exists
581
       if (dest_offset >= 0) {
       // check if the destination file is currently open by any process
for (int i = 0; i < MAX_FDS; i++) {
   if (fd_table[i].in_use && strcmp(fd_table[i].filename, dest) == 0) {</pre>
582
583
584
585
             P_ERRNO = P_EBUSY;
586
             u_perror("mv");
587
             return NULL;
588
           }
        }
589
590
591
         // if destination file exists, delete it
592
        if (mark_entry_as_deleted(&dest_entry, dest_offset) != 0) {
593
           u_perror("mv");
594
           return NULL;
595
596
      }
597
598
599
       strncpy(source_entry.name, dest, sizeof(source_entry.name) - 1);
600
       source_entry.name[sizeof(source_entry.name) - 1] = '\0';
601
      // write the updated entry back to disk
if (lseek(fs_fd, source_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
602
603
604
605
         u_perror("mv");
606
        return NULL;
607
608
609
      if (write(fs_fd, &source_entry, sizeof(source_entry)) != sizeof(source_entry)) {
       P_ERRNO = P_EWRITE;
610
        u_perror("mv");
611
612
         return NULL;
613
      }
614
615
      return NULL;
```

4.2.2.9 rm()

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

Removes files.

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

Parameters

arg | Arguments array (command line arguments)

Returns

void pointer (unused)

```
Definition at line 679 of file fat_routines.c.
```

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

4.2.2.10 touch()

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

Creates empty files or updates timestamps.

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

Parameters

arg | Arguments array (command line arguments)

Returns

void pointer (unused)

Creates empty files or updates timestamps.

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

Definition at line 483 of file fat_routines.c.

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

4.2.2.11 unmount()

```
int unmount ( )
```

Unmounts the currently mounted filesystem.

This function flushes any pending changes and unmounts the filesystem.

Returns

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

Unmounts the currently mounted filesystem.

Definition at line 169 of file fat routines.c.

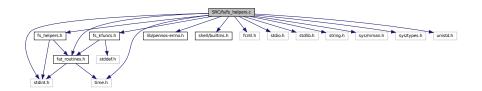
```
169
       // first check that a file system is actually mounted
171
      if (!is_mounted) {
       P_ERRNO = P_EFS_NOT_MOUNTED;
172
173
        return -1;
174
175
176
      // unmap the FAT
177
      if (fat != NULL) {
       if (munmap(fat, fat_size) == -1) {
  P_ERRNO = P_EMAP;
178
179
     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

4.3 SRC/fs/fs_helpers.c File Reference

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

#include <unistd.h>

Include dependency graph for fs_helpers.c:



Functions

• void init_fd_table (fd_entry_t *fd_table)

Initializes the global kernel-level file descriptor table.

int get_free_fd (fd_entry_t *fd_table)

Gets a free file descriptor.

int increment_fd_ref_count (int fd)

Increments the reference count of a file descriptor.

int decrement fd ref count (int fd)

Decrements the reference count of a file descriptor.

int has_executable_permission (int fd)

Checks if a file has executable permissions.

uint16_t allocate_block ()

Allocates a block.

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

Searches for a file in the root directory.

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

Adds a file to the root directory.

• int mark_entry_as_deleted (dir_entry_t *entry, int absolute_offset)

Marks a file entry as deleted and frees its blocks.

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

Copies data from host OS file to the PennFAT file.

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

Copies data from PennFAT file to host OS file.

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

Copies data from source file to destination file.

int compact_directory ()

Compacts a directory.

Variables

- int fs_fd = -1
- int block size = 0
- int num_fat_blocks = 0
- int fat_size = 0
- uint16 t * fat = NULL
- bool is_mounted = false
- int MAX FDS = 100
- fd_entry_t fd_table [100]

4.3.1 Function Documentation

4.3.1.1 add file entry()

Adds a file to the root directory.

Adds a new file entry to the root directory.

```
Definition at line 266 of file fs helpers.c.
```

```
271
      if (!is_mounted) {
        P_ERRNO = P_EFS_NOT_MOUNTED;
return -1;
2.72
273
274
275
276
      // check if file already exists
277
      dir_entry_t existing;
278
      if (find_file(filename, &existing) >= 0) {
      P_ERRNO = P_EEXIST;
return -1;
279
280
281
283
      // start with root directory block (block 1)
284
      uint16_t current_block = 1;
285
      int offset = 0;
      dir_entry_t dir_entry;
286
287
288
      while (1) {
289
        // position at the start of current block of the root directory
290
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
291
          P_ERRNO = P_ELSEEK;
          return -1;
292
293
294
295
        // reset offset for new block
296
        offset = 0;
297
298
        // search current block for free slot
299
        while (offset < block_size) {</pre>
300
          if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
301
            P_ERRNO = P_EREAD;
302
            return -1;
          }
303
304
305
          // found a free slot
          if (dir_entry.name[0] == 0 || dir_entry.name[0] == 1) {
306
307
            // initialize the new entry
308
            memset(&dir_entry, 0, sizeof(dir_entry));
309
             strncpy(dir_entry.name, filename, 31);
            dir_entry.size = size;
dir_entry.firstBlock = first_block;
310
311
            dir_entry type = type;
dir_entry perm = perm;
dir_entry mtime = time(NULL);
312
313
314
315
316
            // write the entry
            if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + offset, SEEK_SET) == -1) {
317
318
             P_ERRNO = P_ELSEEK;
319
              return -1;
320
321
             if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
322
              P_ERRNO = P_EWRITE;
               return -1;
323
324
325
            return offset;
```

```
327
           }
328
329
           offset += sizeof(dir_entry);
        }
330
331
         // current block is full, check if there's a next block
if (fat[current_block] != FAT_EOF) {
332
333
334
          current_block = fat[current_block];
335
           continue;
336
337
         // allocate a new block for the root directory
338
         uint16_t new_block = allocate_block();
if (new_block == 0) {
339
340
341
          P_ERRNO = P_EFULL;
342
           return -1;
343
344
345
         // chain the new block
346
         fat[current_block] = new_block;
347
         fat[new_block] = FAT_EOF;
348
349
         // initialize new block
350
         uint8 t* zero block = calloc(block size, 1);
         if (!zero_block) {
  P_ERRNO = P_EINVAL;
351
352
353
           return -1;
354
355
         // write this new block to the file system
if (lseek(fs_fd, fat_size + (new_block - 1) * block_size, SEEK_SET) == -1) {
356
357
358
           P_ERRNO = P_ELSEEK;
359
           free(zero_block);
360
           return -1;
361
         if (write(fs_fd, zero_block, block_size) != block_size) {
   P_ERRNO = P_EWRITE;
362
363
364
           free(zero_block);
365
           return -1;
366
367
368
         free (zero_block);
369
370
         // initialize the new entry
371
         memset(&dir_entry, 0, sizeof(dir_entry));
372
         strncpy(dir_entry.name, filename, 31);
373
         dir_entry.size = size;
         dir_entry.firstBlock = first_block;
374
375
         dir_entry.type = type;
dir_entry.perm = perm;
dir_entry.mtime = time(NULL);
376
377
378
379
         // write the new entry at the start of the new block in the file system
380
         if (lseek(fs_fd, fat_size + (new_block - 1) * block_size, SEEK_SET) == -1) {
381
          P_ERRNO = P_ELSEEK;
382
           return -1;
383
384
         if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
385
         P_ERRNO = P_EWRITE;
386
           return -1;
387
388
389
         return 0;
390
391 }
```

4.3.1.2 allocate_block()

```
uint16 t allocate block ( )
```

Allocates a block.

Allocates a free block in the FAT.

If no block found, we try compacting the directory.

Definition at line 167 of file fs_helpers.c.

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

4.3.1.3 compact directory()

```
int compact_directory ( )
```

Compacts a directory.

Compacts the root directory by removing all deleted entries.

```
Definition at line 692 of file fs_helpers.c.
```

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

```
739
        }
740
741
         // move onto next block, if there is one
742
        if (fat[current_block] != FAT_EOF) {
743
          current_block = fat[current_block];
744
        } else {
745
          break;
746
747
748
      \ensuremath{//} if no deleted entries, no compaction needed
749
      if (deleted_entries_count == 0) {
750
751
        free(dir_buffer);
752
        return 0;
753
754
755
      \ensuremath{//} allocate space for all valid entries
756
      dir_entry_t* all_entries = malloc(dir_entries_count * sizeof(dir_entry_t));
757
      if (!all_entries) {
758
        P_ERRNO = P_EMALLOC;
759
        free(dir_buffer);
760
        return -1;
761
762
763
      // read all entries into the buffer, skipping deleted ones
764
      current_block = 1;
765
      int valid_entry_idx = 0;
766
767
      while (current_block != FAT_EOF) {
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
768
769
770
           free(dir_buffer);
771
           free(all_entries);
772
          return -1;
773
774
        if (read(fs_fd, dir_buffer, block_size) != block_size) {
   P_ERRNO = P_EREAD;
775
776
777
           free(dir_buffer);
778
          free(all_entries);
779
           return -1;
        }
780
781
782
         // process entries in this block
             (int offset = 0; offset < block_size; offset += sizeof(dir_entry_t)) {</pre>
783
784
           dir_entry_t* entry = (dir_entry_t*) (dir_buffer + offset);
785
           // check if we've reached the end of directory
786
           if (entry->name[0] == 0) {
787
788
            break:
789
           }
790
791
           // skip deleted entries
792
           if (entry->name[0] == 1) {
793
            continue;
794
           }
795
796
           // copy valid entry to our array
797
           memcpy(&all_entries[valid_entry_idx++], entry, sizeof(dir_entry_t));
798
799
800
         // move to the next block
801
        if (fat[current_block] != FAT_EOF) {
          current_block = fat[current_block];
802
803
         } else {
804
          break;
805
        }
806
      }
807
      // rewrite the directory with only valid entries
809
      current_block = 1;
      int entries_per_block = block_size / sizeof(dir_entry_t);
810
      int blocks_needed = (valid_entry_idx + entries_per_block - 1) / entries_per_block;
811
812
      // clean up any excess directory blocks in the FAT chain
813
      uint16_t next_block = fat[current_block];
814
815
      if (blocks_needed == 1) {
        // only need one block, free all others
while (next_block != FAT_EOF) {
  uint16_t temp = fat[next_block];
  fat[next_block] = FAT_FREE;
816
817
818
819
820
          next_block = temp;
821
822
        fat[current_block] = FAT_EOF;
      } else {
  // navigate through needed blocks
823
824
825
        int block count = 1;
```

```
uint16_t prev_block = current_block;
827
828
        while (block_count < blocks_needed) {</pre>
829
          if (next_block == FAT_EOF) {
830
            \ensuremath{//} need to allocate a new block
            uint16_t new_block = allocate_block();
831
            if (new_block == 0) {
832
833
               P_ERRNO = P_EFULL;
834
              free(dir_buffer);
835
              free(all_entries);
836
              return -1;
837
838
             fat[prev_block] = new_block;
839
            next_block = new_block;
840
841
          prev_block = next_block;
next_block = fat[next_block];
842
843
844
          block_count++;
845
846
847
        // free any excess blocks
848
        fat[prev_block] = FAT_EOF;
        while (next_block != FAT_EOF) {
849
850
          uint16_t temp = fat[next_block];
          fat[next_block] = FAT_FREE;
851
852
          next_block = temp;
853
854
855
856
      // write the valid entries back to the directory blocks
857
      current_block = 1;
858
      int entries_written = 0;
859
860
      while (entries_written < valid_entry_idx) {</pre>
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
861
862
          free(dir_buffer);
863
864
          free(all_entries);
865
          return -1;
866
867
        memset(dir_buffer, 0, block_size);
868
869
870
        // copy entries to the buffer
871
        int entries_in_this_block = 0;
872
        while (entries_written < valid_entry_idx && entries_in_this_block < entries_per_block) {</pre>
873
          memcpy(dir_buffer + (entries_in_this_block * sizeof(dir_entry_t)),
874
                  &all_entries[entries_written],
875
                   sizeof(dir_entry_t));
          entries_written++;
877
          entries_in_this_block++;
878
879
880
        // write the buffer to the file system
        if (write(fs_fd, dir_buffer, block_size) != block_size) {
   P_ERRNO = P_EINVAL;
881
883
          free(dir_buffer);
884
          free(all_entries);
885
          return -1;
886
887
888
        // move to the next block if needed
889
        if (entries_written < valid_entry_idx) {</pre>
890
          current_block = fat[current_block];
891
892
893
894
      free (dir_buffer);
      free(all_entries);
896
      return 0;
897 }
```

4.3.1.4 copy_host_to_pennfat()

Copies data from host OS file to the PennFAT file.

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

Definition at line 434 of file fs_helpers.c.

```
435
436
       if (!is mounted) {
        P_ERRNO = P_EFS_NOT_MOUNTED;
437
        return -1;
438
439
440
441
      // open the host file
      int host_fd = open(host_filename, O_RDONLY);
if (host_fd == -1) {
442
443
       P_ERRNO = P_EOPEN;
444
445
        return -1;
446
447
      // determine file size by seeking to the end and getting position
off_t host_file_size_in_bytes = lseek(host_fd, 0, SEEK_END);
if (host_file_size_in_bytes == -1) {
448
449
450
        P_ERRNO = P_ELSEEK;
451
452
        close(host_fd);
453
         return -1;
454
455
      // go back to beginning of file for reading
456
      if (lseek(host_fd, 0, SEEK_SET) == -1) {
457
458
        P_ERRNO = P_ELSEEK;
459
       close(host_fd);
460
        return -1;
461
462
463
      // open the destination file in PennFAT
      int pennfat_fd = k_open(pennfat_filename, F_WRITE);
if (pennfat_fd < 0) {</pre>
464
465
466
       close(host_fd);
467
        return -1;
468
469
470
       // copy the data into this buffer
471
      uint8_t* buffer = (uint8_t*)malloc(block_size);
      if (!buffer) {
  P_ERRNO = P_EMALLOC;
472
473
474
        k close (pennfat fd);
475
        close (host_fd);
476
        return -1;
477
478
      uint32_t bytes_remaining = host_file_size_in_bytes;
479
480
      ssize_t bytes_read;
481
482
      // read from host file
483
      while (bytes_remaining > 0) {
484
        // ensure bytes to read never exceeds the block size
        ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
485
486
        bytes_read = read(host_fd, buffer, bytes_to_read);
487
488
        if (bytes_read <= 0) {</pre>
489
          break;
490
491
        // write to pennfat_fd using k\_write
492
        if (k_write(pennfat_fd, (const char*)buffer, bytes_read) != bytes_read) {
493
494
          free (buffer);
495
           k_close (pennfat_fd);
496
           close(host_fd);
497
          return -1;
498
499
500
        bytes_remaining -= bytes_read;
501
502
503
       // check for read error
      if (bytes_read < 0) {
  P_ERRNO = P_EREAD;</pre>
504
505
        free (buffer);
506
507
        k_close (pennfat_fd);
508
        close(host_fd);
509
        return -1;
510
511
      // otherwise, cleanup and return success
512
513
      free (buffer);
      k_close(pennfat_fd);
```

```
515    close(host_fd);
516    return 0;
517 }
```

4.3.1.5 copy_pennfat_to_host()

Copies data from PennFAT file to host OS file.

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

Definition at line 522 of file fs_helpers.c.

```
523
524
      if (!is mounted) {
525
        P_ERRNO = P_EFS_NOT_MOUNTED;
526
        return -1;
527
528
529
      // open the PennFAT file
      int pennfat_fd = k_open(pennfat_filename, F_READ);
if (pennfat_fd < 0) {</pre>
530
531
532
        return -1;
533
534
535
      \ensuremath{//} get the pennfat file size
      off_t pennfat_file_size_in_bytes = k_lseek(pennfat_fd, 0, SEEK_END);
if (pennfat_file_size_in_bytes == -1) {
536
537
       k_close(pennfat_fd);
538
539
        return -1;
540
541
      // go back to beginning of file for reading
if (k_lseek(pennfat_fd, 0, SEEK_SET) == -1) {
542
543
544
       k_close(pennfat_fd);
545
        return -1;
546
547
548
      // open the host file
      int host_fd = open(host_filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
549
      if (host_fd == -1) {
550
       P_ERRNO = P_EOPEN;
551
552
        k_close(pennfat_fd);
553
554
555
556
      // allocate buffer for data transfer
557
      char* buffer = (char*)malloc(block_size);
      if (!buffer) {
        P_ERRNO = P_EMALLOC;
559
560
        k_close(pennfat_fd);
561
        close(host_fd);
        return -1;
562
563
564
565
      uint32_t bytes_remaining = pennfat_file_size_in_bytes;
566
      ssize_t bytes_read;
567
      // read from PennFAT file and write to host file
568
569
      while (bytes_remaining > 0) {
570
        // ensure bytes to read never exceeds the block size
571
         ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
572
        bytes_read = k_read(pennfat_fd, buffer, bytes_to_read);
573
574
        if (bytes_read <= 0) {</pre>
575
          break;
576
577
578
        if (write(host_fd, buffer, bytes_read) != bytes_read) {
579
         P_ERRNO = P_EINVAL;
          free (buffer);
580
581
          close(host fd);
582
          k_close (pennfat_fd);
          return -1;
```

```
584
       }
585
586
       bytes_remaining -= bytes_read;
587
588
      // check for read error
589
     if (bytes_read < 0) {</pre>
590
591
        P_ERRNO = P_EREAD;
592
        free (buffer);
593
        close(host_fd);
594
       k_close(pennfat_fd);
595
       return -1;
596
597
598
     // otherwise, cleanup and return success
599
     free (buffer);
600
      close(host_fd);
601
     k_close (pennfat_fd);
     return 0;
602
603 }
```

4.3.1.6 copy source to dest()

Copies data from source file to destination file.

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

Definition at line 608 of file fs_helpers.c.

```
610
      if (!is_mounted) {
        P_ERRNO = P_EFS_NOT_MOUNTED;
611
        return -1;
612
613
614
615
      // open the source file
      int source_fd = k_open(source_filename, F_READ);
if (source_fd < 0) {</pre>
616
617
      return -1;
}
618
619
620
621
      // get the source file size
      off_t source_file_size_in_bytes = k_lseek(source_fd, 0, SEEK_END);
if (source_file_size_in_bytes == -1) {
622
623
624
       k_close(source_fd);
625
        return -1;
626
627
628
      \ensuremath{//} move to the beginning of the source file for reading
629
      if (k_lseek(source_fd, 0, SEEK_SET) < 0) {</pre>
630
       k_close(source_fd);
631
        return -1;
632
633
634
      \ensuremath{//} open the destination file
635
      int dest_fd = k_open(dest_filename, F_WRITE);
      if (dest_fd < 0) {
636
637
        k_close(source_fd);
638
        return -1;
639
640
641
      \ensuremath{//} read from source to destination
642
      char* buffer = (char*)malloc(block_size);
      if (!buffer) {
  P_ERRNO = P_EMALLOC;
643
644
645
        k_close(source_fd);
646
        k_close(dest_fd);
647
648
649
650
      uint32 t bytes remaining = source file size in bytes;
651
      ssize_t bytes_read;
```

```
while (bytes_remaining > 0) {
       // make sure the bytes to read doesn't exceed block size
ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
654
655
        bytes_read = k_read(source_fd, buffer, bytes_to_read);
656
657
658
       _ (byte
   break;
}
        if (bytes_read <= 0) {</pre>
660
661
662
        if (k_write(dest_fd, buffer, bytes_read) != bytes_read) {
         free(buffer);
663
          k close(source fd);
664
665
          k_close(dest_fd);
666
          return -1;
667
        }
668 }
669
      // check for read error
670
671
      if (bytes_read < 0) {</pre>
672
       free (buffer);
673
        k_close(source_fd);
674
        k_close(dest_fd);
675
        return -1;
676
677
678
     // otherwise, cleanup and return success
679
      free (buffer);
680
      k_close(source_fd);
681
      k_close(dest_fd);
682
     return 0;
683 }
```

4.3.1.7 decrement fd ref count()

Decrements the reference count of a file descriptor.

If reference count reaches 0, flush field values.

```
Definition at line 107 of file fs helpers.c.
```

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

4.3.1.8 find_file()

Searches for a file in the root directory.

Retrieves the file's absolute offset in the filesystem.

Definition at line 192 of file fs_helpers.c.

```
192
193
       if (!is mounted) {
194
        P_ERRNO = P_EFS_NOT_MOUNTED;
        return -1;
195
196
197
198
      // Start with root directory block (block 1)
199
      uint16_t current_block = 1;
      int offset_in_block = 0;
int absolute_offset = 0;
dir_entry_t dir_entry;
200
201
202
203
204
205
        // Position at the start of current block
206
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
207
          P_ERRNO = P_ELSEEK;
208
          return -1;
209
210
211
         // reset offset for new block
212
        offset_in_block = 0;
213
214
        // calculate the absolute offset
215
        absolute_offset = fat_size + (current_block - 1) * block_size;
216
217
         // search current block
218
        while (offset_in_block < block_size) {</pre>
          if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
   P_ERRNO = P_EREAD;
219
220
221
            return -1;
222
223
224
           // check if we've reached the end of directory
225
           if (dir_entry.name[0] == 0) {
226
            break;
227
229
           // check if this is a deleted entry
230
           if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
            offset_in_block += sizeof(dir_entry);
absolute_offset += sizeof(dir_entry);
231
232
233
             continue:
234
235
236
           // check if we found the file
237
           if (strcmp(dir_entry.name, filename) == 0) {
238
             if (entry) {
239
              memcpy(entry, &dir_entry, sizeof(dir_entry));
241
             return absolute_offset; // return the absolute file offset
242
243
          offset_in_block += sizeof(dir_entry);
absolute_offset += sizeof(dir_entry);
2.44
245
246
247
248
         // if we've reached the end of the current block, check if there's a next block
249
         if (fat[current_block] != FAT_EOF) {
250
          current_block = fat[current_block];
251
          continue;
252
253
         // no more blocks to search
254
255
256
257
      // file not found
258
     P_ERRNO = P_ENOENT;
260
      return -1;
261 }
```

4.3.1.9 get_free_fd()

Gets a free file descriptor.

Finds the first available file descriptor in the table.

Definition at line 77 of file fs_helpers.c.

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

4.3.1.10 has_executable_permission()

Checks if a file has executable permissions.

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

Definition at line 133 of file fs_helpers.c.

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

4.3.1.11 increment_fd_ref_count()

Increments the reference count of a file descriptor.

Parameters

fd | file descriptor to increment

Returns

new reference count, or -1 on error

Definition at line 89 of file fs_helpers.c.

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

4.3.1.12 init_fd_table()

Initializes the global kernel-level file descriptor table.

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

Definition at line 43 of file fs_helpers.c.

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

4.3.1.13 mark_entry_as_deleted()

```
int mark\_entry\_as\_deleted (
             dir_entry_t * entry,
             int absolute_offset )
```

Marks a file entry as deleted and frees its blocks.

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

Definition at line 396 of file fs helpers.c.

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

4.3.2 Variable Documentation

4.3.2.1 block size

```
int block\_size = 0
```

Definition at line 28 of file fs_helpers.c.

4.3.2.2 fat

```
uint16_t* fat = NULL
```

Definition at line 31 of file fs_helpers.c.

4.3.2.3 fat_size

```
int fat_size = 0
```

Definition at line 30 of file fs_helpers.c.

4.3.2.4 fd_table

```
fd_entry_t fd_table[100]
```

Definition at line 34 of file fs_helpers.c.

4.3.2.5 fs_fd

```
int fs_fd = -1
```

Definition at line 27 of file fs_helpers.c.

4.3.2.6 is_mounted

```
bool is_mounted = false
```

Definition at line 32 of file fs_helpers.c.

4.3.2.7 MAX FDS

```
int MAX_FDS = 100
```

Definition at line 33 of file fs_helpers.c.

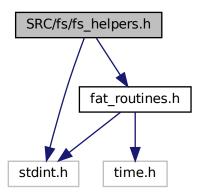
4.3.2.8 num_fat_blocks

```
int num_fat_blocks = 0
```

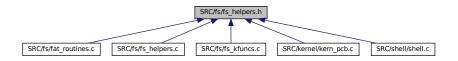
Definition at line 29 of file fs_helpers.c.

4.4 SRC/fs/fs_helpers.h File Reference

```
#include <stdint.h>
#include "fat_routines.h"
Include dependency graph for fs_helpers.h:
```



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



Functions

void init_fd_table (fd_entry_t *fd_table)

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

int get_free_fd (fd_entry_t *fd_table)

Finds the first available file descriptor in the table.

int increment_fd_ref_count (int fd)

Increments the reference count of a file descriptor.

int decrement_fd_ref_count (int fd)

Decrements the reference count of a file descriptor.

• int has_executable_permission (int fd)

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

uint16_t allocate_block ()

Allocates a free block in the FAT.

int find file (const char *filename, dir entry t *entry)

Searches for a file in the root directory.

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

Adds a new file entry to the root directory.

int mark_entry_as_deleted (dir_entry_t *entry, int offset)

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

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

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

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

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

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

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

• int compact_directory ()

Compacts the root directory by removing all deleted entries.

Variables

- int fs fd
- · int block size
- int num_fat_blocks
- int fat_size
- uint16_t * fat
- bool is mounted
- int MAX FDS
- fd_entry_t fd_table [100]

4.4.1 Function Documentation

4.4.1.1 add_file_entry()

Adds a new file entry to the root directory.

Parameters

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

Returns

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

Adds a new file entry to the root directory.

```
Definition at line 266 of file fs helpers.c.
```

```
270
      if (!is_mounted) {
272
       P_ERRNO = P_EFS_NOT_MOUNTED;
273
        return -1;
274
275
      // check if file already exists
dir_entry_t existing;
276
      if (find_file(filename, &existing) >= 0) {
279
      P_ERRNO = P_EEXIST;
280
        return -1;
281
282
      // start with root directory block (block 1)
uint16_t current_block = 1;
283
285
      int offset = 0;
286
      dir_entry_t dir_entry;
287
288
      while (1) {
289
       // position at the start of current block of the root directory
290
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
291
         P_ERRNO = P_ELSEEK;
292
           return -1;
293
        }
2.94
        // reset offset for new block
295
        offset = 0;
296
297
298
         // search current block for free slot
299
        while (offset < block_size) {</pre>
          if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
300
            P_ERRNO = P_EREAD;
301
302
            return -1;
303
304
305
           // found a free slot
           if (dir_entry.name[0] == 0 || dir_entry.name[0] == 1) {
306
            // initialize the new entry memset(&dir_entry, 0, sizeof(dir_entry));
307
308
309
             strncpy(dir_entry.name, filename, 31);
310
             dir_entry.size = size;
311
             dir_entry.firstBlock = first_block;
            dir_entry.type = type;
dir_entry.perm = perm;
312
313
            dir_entry.mtime = time(NULL);
314
315
316
             // write the entry
317
             if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + offset, SEEK_SET) == -1) {
318
              P_ERRNO = P_ELSEEK;
               return -1;
319
320
321
             if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
322
               P_ERRNO = P_EWRITE;
323
               return -1;
324
325
326
             return offset;
327
328
329
          offset += sizeof(dir_entry);
330
331
        // current block is full, check if there's a next block
if (fat[current_block] != FAT_EOF) {
332
333
334
         current_block = fat[current_block];
335
336
337
338
        // allocate a new block for the root directory
339
        uint16 t new block = allocate block();
        if (new_block == 0) {
340
341
         P_ERRNO = P_EFULL;
342
          return -1;
343
344
         // chain the new block
345
        fat[current_block] = new_block;
```

```
fat[new_block] = FAT_EOF;
347
348
349
         // initialize new block
         uint8_t* zero_block = calloc(block_size, 1);
350
351
         if (!zero_block) {
  P_ERRNO = P_EINVAL;
352
353
           return -1;
354
355
         // write this new block to the file system
if (lseek(fs_fd, fat_size + (new_block - 1) * block_size, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
356
357
358
359
           free(zero_block);
360
           return -1;
361
362
         if (write(fs_fd, zero_block, block_size) != block_size) {
           P ERRNO = P EWRITE:
363
           free(zero_block);
364
365
           return -1;
366
367
368
         free(zero_block);
369
        // initialize the new entry
memset(&dir_entry, 0, sizeof(dir_entry));
strncpy(dir_entry.name, filename, 31);
dir_entry.size = size;
370
371
372
373
374
         dir_entry.firstBlock = first_block;
375
         dir_entry.type = type;
         dir_entry.perm = perm;
376
377
         dir_entry.mtime = time(NULL);
378
379
         // write the new entry at the start of the new block in the file system
380
         if (lseek(fs_fd, fat_size + (new_block - 1) * block_size, SEEK_SET) == -1) {
          P_ERRNO = P_ELSEEK;
return -1;
381
382
383
384
         if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
385
          P_ERRNO = P_EWRITE;
386
           return -1;
387
388
389
         return 0;
390
      }
391 }
```

4.4.1.2 allocate block()

```
uint16_t allocate_block ( )
```

Allocates a free block in the FAT.

Returns

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

Allocates a free block in the FAT.

If no block found, we try compacting the directory.

Definition at line 167 of file fs_helpers.c.

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

```
179 return i;

180 }

181 }

182 }

183 

184 return 0;

185 }
```

4.4.1.3 compact directory()

```
int compact_directory ( )
```

Compacts the root directory by removing all deleted entries.

Returns

0 on success, -1 on error

Compacts the root directory by removing all deleted entries.

Definition at line 692 of file fs helpers.c.

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

```
745
                              break;
746
747
748
                   // if no deleted entries, no compaction needed
if (deleted_entries_count == 0) {
749
750
751
                    free(dir_buffer);
752
753
754
755
                   // allocate space for all valid entries
                   dir_entry_t* all_entries = malloc(dir_entries_count * sizeof(dir_entry_t));
756
 757
                   if (!all_entries) {
758
                      P_ERRNO = P_EMALLOC;
759
                         free(dir_buffer);
760
                         return -1;
761
762
763
                   // read all entries into the buffer, skipping deleted ones
 764
                   current_block = 1;
765
                   int valid_entry_idx = 0;
766
767
                   while (current_block != FAT_EOF) {
                       if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
768
769
770
                                free(dir_buffer);
                                free(all_entries);
771
772
                               return -1;
773
774
775
                         if (read(fs_fd, dir_buffer, block_size) != block_size) {
 776
                               P_ERRNO = P_EREAD;
777
                                free(dir_buffer);
778
                                free(all_entries);
779
                               return -1;
780
781
782
                          // process entries in this block
783
                                      (int offset = 0; offset < block_size; offset += sizeof(dir_entry_t)) {</pre>
784
                               dir_entry_t* entry = (dir_entry_t*) (dir_buffer + offset);
785
                                // check if we've reached the end of directory % \left( 1\right) =\left( 1\right) \left( 1\right)
786
                                if (entry->name[0] == 0) {
787
788
                                    break;
789
790
791
                                // skip deleted entries
792
                                if (entry->name[0] == 1) {
793
                                    continue:
794
795
796
                                 // copy valid entry to our array
797
                                memcpy(&all_entries[valid_entry_idx++], entry, sizeof(dir_entry_t));
798
799
800
                         // move to the next block
                         if (fat[current_block] != FAT_EOF) {
801
802
                              current_block = fat[current_block];
803
                          } else {
804
                              break;
805
                        }
806
807
                   \ensuremath{//} rewrite the directory with only valid entries
809
                   current_block = 1;
810
                   int entries_per_block = block_size / sizeof(dir_entry_t);
                   int blocks_needed = (valid_entry_idx + entries_per_block - 1) / entries_per_block;
811
812
813
                    // clean up any excess directory blocks in the FAT chain
                   if (blocks_needed == 1) {
814
815
816
                          // only need one block, free all others
                         while (next_block != FAT_EOF) {
817
                              uint16_t temp = fat[next_block];
fat[next_block] = FAT_FREE;
818
819
820
                               next_block = temp;
821
822
                         fat[current_block] = FAT_EOF;
823
                   } else {
824
                         // navigate through needed blocks
825
                         int block count = 1;
826
                         uint16_t prev_block = current_block;
827
828
                         while (block_count < blocks_needed) {</pre>
829
                            if (next_block == FAT_EOF) {
                                      // need to allocate a new block
uint16_t new_block = allocate_block();
830
831
```

```
832
             if (new_block == 0) {
833
               P_ERRNO = P_EFULL;
834
               free(dir_buffer);
835
               free(all_entries);
836
               return -1;
837
             fat[prev_block] = new_block;
838
839
             next_block = new_block;
840
841
          prev_block = next_block;
next_block = fat[next_block];
842
843
844
          block_count++;
845
846
        // free any excess blocks
fat[prev_block] = FAT_EOF;
while (next_block != FAT_EOF) {
847
848
849
         uint16_t temp = fat[next_block];
850
851
           fat[next_block] = FAT_FREE;
852
          next_block = temp;
853
854
      }
855
      // write the valid entries back to the directory blocks
856
      current_block = 1;
858
      int entries_written = 0;
859
860
      while (entries_written < valid_entry_idx) {</pre>
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
861
862
863
           free(dir_buffer);
864
           free(all_entries);
865
           return -1;
866
867
        memset(dir buffer, 0, block size);
868
869
870
        // copy entries to the buffer
871
        int entries_in_this_block = 0;
872
        while (entries_written < valid_entry_idx && entries_in_this_block < entries_per_block) {</pre>
873
          memcpy(dir_buffer + (entries_in_this_block * sizeof(dir_entry_t)),
874
                   &all entries[entries written],
875
                   sizeof(dir_entry_t));
876
          entries_written++;
877
          entries_in_this_block++;
878
879
880
        // write the buffer to the file system
        if (write(fs_fd, dir_buffer, block_size) != block_size) {
881
          P_ERRNO = P_EINVAL;
882
883
          free(dir_buffer);
884
           free(all_entries);
885
          return -1;
886
887
        // move to the next block if needed
888
889
        if (entries_written < valid_entry_idx) {</pre>
890
          current_block = fat[current_block];
891
892
      }
893
894
      free(dir_buffer);
895
      free(all_entries);
896
      return 0;
897 }
```

4.4.1.4 copy_host_to_pennfat()

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

Parameters

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

Returns

0 on success, -1 on error

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

Definition at line 434 of file fs_helpers.c.

```
435
       if (!is mounted) {
436
437
        P_ERRNO = P_EFS_NOT_MOUNTED;
438
         return -1;
439
440
      \ensuremath{//} open the host file
441
      int host_fd = open(host_filename, O_RDONLY);
if (host_fd == -1) {
442
443
       P_ERRNO = P_EOPEN;
444
445
        return -1;
446
447
      // determine file size by seeking to the end and getting position
448
      off_t host_file_size_in_bytes = lseek(host_fd, 0, SEEK_END);
if (host_file_size_in_bytes == -1) {
   P_ERRNO = P_ELSEEK;
449
450
451
452
         close(host_fd);
453
         return -1;
454
455
456
      // go back to beginning of file for reading
457
      if (lseek(host_fd, 0, SEEK_SET) == -1) {
458
        P_ERRNO = P_ELSEEK;
459
         close(host_fd);
460
         return -1;
461
462
463
       // open the destination file in PennFAT
      int pennfat_fd = k_open(pennfat_filename, F_WRITE);
if (pennfat_fd < 0) {</pre>
464
465
466
       close(host_fd);
467
         return -1;
468
469
470
       // copy the data into this buffer
471
       uint8_t* buffer = (uint8_t*)malloc(block_size);
      if (!buffer) {
  P_ERRNO = P_EMALLOC;
  k_close(pennfat_fd);
472
473
474
475
        close (host_fd);
476
        return -1;
477
478
479
      uint32_t bytes_remaining = host_file_size_in_bytes;
480
      ssize_t bytes_read;
481
482
       // read from host file
483
       while (bytes_remaining > 0) {
484
        // ensure bytes to read never exceeds the block size
        ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;
bytes_read = read(host_fd, buffer, bytes_to_read);
485
486
487
         if (bytes_read <= 0) {</pre>
488
489
           break;
490
         }
491
492
         // write to pennfat_fd using k_write
         if (k_write(pennfat_fd, (const char*)buffer, bytes_read) != bytes_read) {
493
494
          free (buffer);
495
           k_close (pennfat_fd);
496
           close(host_fd);
497
           return -1;
498
499
500
         bytes_remaining -= bytes_read;
```

```
502
503
      // check for read error
504
      if (bytes_read < 0) {</pre>
      P_ERRNO = P_EREAD;
505
       free(buffer);
506
       k_close(pennfat_fd);
507
508
       close(host_fd);
509
        return -1;
510
511
     // otherwise, cleanup and return success
512
513
     free (buffer);
     k_close(pennfat_fd);
514
515
     close(host_fd);
516
     return 0;
517 }
```

4.4.1.5 copy_pennfat_to_host()

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

Parameters

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

Returns

0 on success, -1 on error

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

Definition at line 522 of file fs_helpers.c.

```
523
       if (!is_mounted) {
   P_ERRNO = P_EFS_NOT_MOUNTED;
524
525
526
         return -1;
527
528
529
       // open the PennFAT file
530
       int pennfat_fd = k_open(pennfat_filename, F_READ);
if (pennfat_fd < 0) {</pre>
531
532
         return -1;
533
534
535
       \ensuremath{//} get the pennfat file size
       off_t pennfat_file_size_in_bytes = k_lseek(pennfat_fd, 0, SEEK_END);
if (pennfat_file_size_in_bytes == -1) {
536
537
538
        k_close(pennfat_fd);
539
         return -1;
540
541
       // go back to beginning of file for reading
if (k_lseek(pennfat_fd, 0, SEEK_SET) == -1) {
542
543
        k_close(pennfat_fd);
return -1;
544
545
546
547
548
       \ensuremath{//} open the host file
       int host_fd = open(host_filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (host_fd == -1) {
549
550
551
         P_ERRNO = P_EOPEN;
          k_close (pennfat_fd);
```

```
553
        return -1;
554
555
      \ensuremath{//} allocate buffer for data transfer
556
557
      char* buffer = (char*)malloc(block_size);
if (!buffer) {
558
       P_ERRNO = P_EMALLOC;
559
560
        k_close(pennfat_fd);
561
       close(host_fd);
562
        return -1;
563
564
565
      uint32_t bytes_remaining = pennfat_file_size_in_bytes;
566
      ssize_t bytes_read;
567
568
      // read from PennFAT file and write to host file
569
      while (bytes_remaining > 0) {
        // ensure bytes to read never exceeds the block size
ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
570
571
572
        bytes_read = k_read(pennfat_fd, buffer, bytes_to_read);
573
574
        if (bytes_read <= 0) {</pre>
575
          break;
576
577
578
        if (write(host_fd, buffer, bytes_read) != bytes_read) {
          P_ERRNO = P_EINVAL;
579
580
          free (buffer);
581
          close(host_fd);
582
          k_close(pennfat_fd);
583
          return -1;
584
585
586
        bytes_remaining -= bytes_read;
587
588
589
      // check for read error
      if (bytes_read < 0) {
   P_ERRNO = P_EREAD;</pre>
590
591
592
        free (buffer);
593
        close(host_fd);
594
        k_close(pennfat_fd);
595
        return -1;
596
597
598
      // otherwise, cleanup and return success
599
      free (buffer);
600
      close(host_fd);
     k_close(pennfat_fd);
return 0;
601
602
603 }
```

4.4.1.6 copy_source_to_dest()

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

Parameters

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

Returns

0 on success, -1 on error

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

```
Definition at line 608 of file fs_helpers.c.
610
      if (!is_mounted) {
        P_ERRNO = P_EFS_NOT_MOUNTED;
611
        return -1;
612
613
614
615
      \ensuremath{//} open the source file
616
      int source_fd = k_open(source_filename, F_READ);
      if (source_fd < 0) {</pre>
617
618
        return -1;
619
620
621
      // get the source file size
622
      off_t source_file_size_in_bytes = k_lseek(source_fd, 0, SEEK_END);
      if (source_file_size_in_bytes == -1) {
623
624
        k_close(source_fd);
625
        return -1;
626
627
628
      \ensuremath{//} move to the beginning of the source file for reading
629
      if (k_lseek(source_fd, 0, SEEK_SET) < 0) {</pre>
      k_close(source_fd);
630
631
        return -1;
632
633
634
      // open the destination file
635
      int dest_fd = k_open(dest_filename, F_WRITE);
      if (dest_fd < 0) {
636
        k_close(source_fd);
637
638
        return -1:
639
640
641
      \ensuremath{//} read from source to destination
642
      char* buffer = (char*)malloc(block_size);
643
      if (!buffer) {
       P_ERRNO = P_EMALLOC;
644
645
        k_close(source_fd);
646
        k_close(dest_fd);
647
        return -1;
648
649
650
      uint32_t bytes_remaining = source_file_size_in_bytes;
651
      ssize_t bytes_read;
653
      while (bytes_remaining > 0) {
       // make sure the bytes to read doesn't exceed block size
ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
654
655
656
        bytes_read = k_read(source_fd, buffer, bytes_to_read);
657
658
        if (bytes_read <= 0) {</pre>
659
         break;
660
661
        if (k_write(dest_fd, buffer, bytes_read) != bytes_read) {
662
         free (buffer);
663
664
           k_close(source_fd);
665
          k_close(dest_fd);
666
          return -1;
667
        }
668
     }
669
670
      // check for read error
671
      if (bytes_read < 0) {</pre>
672
        free (buffer);
673
        k_close(source_fd);
674
        k_close(dest_fd);
675
        return -1;
676
```

4.4.1.7 decrement_fd_ref_count()

free (buffer);

return 0;

k_close(source_fd);

k_close(dest_fd);

// otherwise, cleanup and return success

678

679

680

681

682

Decrements the reference count of a file descriptor.

Parameters

```
fd file descriptor to decrement
```

Returns

new reference count, or -1 on error

If reference count reaches 0, flush field values.

Definition at line 107 of file fs_helpers.c.

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

4.4.1.8 find_file()

Searches for a file in the root directory.

Parameters

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

Returns

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

Retrieves the file's absolute offset in the filesystem.

Definition at line 192 of file fs helpers.c.

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

```
194
        P_ERRNO = P_EFS_NOT_MOUNTED;
195
        return -1;
196
197
      // Start with root directory block (block 1)
198
      uint16_t current_block = 1;
199
      int offset_in_block = 0;
200
201
      int absolute_offset = 0;
202
      dir_entry_t dir_entry;
203
204
      while (1) {
        // Position at the start of current block
205
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
206
207
         P_ERRNO = P_ELSEEK;
208
           return -1;
209
210
211
        // reset offset for new block
212
        offset_in_block = 0;
213
214
         // calculate the absolute offset
215
        absolute_offset = fat_size + (current_block - 1) * block_size;
216
217
        // search current block
218
        while (offset_in_block < block_size) {</pre>
         if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
219
220
            P_ERRNO = P_EREAD;
221
            return -1;
222
223
224
          // check if we've reached the end of directory
225
           if (dir_entry.name[0] == 0) {
226
            break;
227
228
          // check if this is a deleted entry
if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
  offset_in_block += sizeof(dir_entry);
229
230
231
232
            absolute_offset += sizeof(dir_entry);
233
            continue;
234
235
          // check if we found the file
236
          if (strcmp(dir_entry.name, filename) == 0) {
237
            if (entry) {
239
              memcpy(entry, &dir_entry, sizeof(dir_entry));
240
             return absolute_offset; // return the absolute file offset
2.41
          }
242
243
244
          offset_in_block += sizeof(dir_entry);
245
          absolute_offset += sizeof(dir_entry);
246
2.47
        // if we've reached the end of the current block, check if there's a next block
if (fat[current_block] != FAT_EOF) {
248
249
250
         current_block = fat[current_block];
251
          continue;
252
253
        \ensuremath{//} no more blocks to search
254
255
        break;
256
257
258
      // file not found
259
     P_ERRNO = P_ENOENT;
260
      return -1;
261 }
```

4.4.1.9 get_free_fd()

Finds the first available file descriptor in the table.

Parameters

fd_table pointer to the file descriptor table to search

Returns

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

Finds the first available file descriptor in the table.

Definition at line 77 of file fs helpers.c.

4.4.1.10 has_executable_permission()

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

Parameters

fd The fd of the file to check.

Returns

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

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

Definition at line 133 of file fs_helpers.c.

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

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

4.4.1.11 increment fd ref count()

Increments the reference count of a file descriptor.

Parameters

fd | file descriptor to increment

Returns

new reference count, or -1 on error

Definition at line 89 of file fs_helpers.c.

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

4.4.1.12 init_fd_table()

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

Parameters

fd_table | pointer to the file descriptor table to initialize

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

Definition at line 43 of file fs_helpers.c.

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

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

4.4.1.13 mark_entry_as_deleted()

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

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

Parameters

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

Returns

0 on success, -1 on error

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

Definition at line 396 of file fs_helpers.c.

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

```
411
       dir_entry_t deleted_entry = *entry;
       deleted_entry.name[0] = 1;
if (lseek(fs_fd, absolute_offset, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
    return -1;
412
413
414
415
416
417
       if (write(fs_fd, &deleted_entry, sizeof(deleted_entry)) != sizeof(deleted_entry)) {
       P_ERRNO = P_EINVAL;
return -1;
418
419
420
421
// mark the passed entry as deleted
423 entry->name[0] = 1;
424 return 0;
425 }
```

4.4.2 Variable Documentation

4.4.2.1 block size

```
int block_size [extern]
```

Definition at line 28 of file fs_helpers.c.

4.4.2.2 fat

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

Definition at line 31 of file fs_helpers.c.

4.4.2.3 fat_size

```
int fat_size [extern]
```

Definition at line 30 of file fs_helpers.c.

4.4.2.4 fd_table

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

Definition at line 34 of file fs_helpers.c.

4.4.2.5 fs_fd

```
int fs_fd [extern]
```

Definition at line 27 of file fs_helpers.c.

4.4.2.6 is_mounted

```
bool is_mounted [extern]
```

Definition at line 32 of file fs_helpers.c.

4.4.2.7 MAX FDS

```
int MAX_FDS [extern]
```

Definition at line 33 of file fs_helpers.c.

4.4.2.8 num_fat_blocks

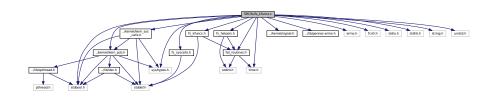
```
int num_fat_blocks [extern]
```

Definition at line 29 of file fs_helpers.c.

4.5 SRC/fs/fs_kfuncs.c File Reference

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

Include dependency graph for fs_kfuncs.c:



Functions

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

Kernel-level call to open a file.

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

Kernel-level call to read a file.

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

Kernel-level call to write to a file.

• int k_close (int fd)

Kernel-level call to close a file.

• int k_unlink (const char *fname)

Kernel-level call to remove a file.

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

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

- void format_file_info (dir_entry_t *entry, char *buffer)
- int k ls (const char *filename)

Kernel-level call to list files.

Variables

- pcb_t * current_running_pcb
- pid_t current_fg_pid

4.5.1 Function Documentation

4.5.1.1 format_file_info()

Definition at line 666 of file fs kfuncs.c.

```
666
667
         // convert permissions to string
668
         char perms[4] = "---";
        cnar perms[4] = "---";
if (entry->perm & PERM_READ)
  perms[0] = 'r';
if (entry->perm & PERM_WRITE)
  perms[1] = 'w';
if (entry->perm & PERM_READ_EXEC & ~PERM_READ)
  perms[2] = 'x';
669
670
671
672
673
674
675
        // convert time to string
676
        char time_str[20];
struct tm* tm = localtime(&entry->mtime);
677
678
        strftime(time_str, sizeof(time_str), "%b %d %H:%M", tm);
680
681 // format the output string
682 snprintf(buffer, 256, "%4d %s %6d %s %s\n", entry->firstBlock, perms,
683 entry->size, time_str, entry->name);
684 }
```

4.5.1.2 k_close()

```
int k\_close ( int fd)
```

Kernel-level call to close a file.

Closes an open file.

Definition at line 528 of file fs kfuncs.c.

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

4.5.1.3 k_ls()

Kernel-level call to list files.

Lists files or file information.

Definition at line 689 of file fs_kfuncs.c.

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

```
fat_size + (current_block - 1) * block_size + offset_in_block;
706
707
           // read directory entry
           if (lseek(fs_fd, abs_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
708
709
710
             return -1;
711
712
           if (read(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
713
           P_ERRNO = P_EREAD;
714
             return -1;
715
           }
716
717
           // check for end of directory
718
           if (entry.name[0] == 0)
719
720
721
           // skip deleted entries
           if (entry.name[0] == 1 || entry.name[0] == 2) {
  offset_in_block += sizeof(entry);
722
723
724
             // check if we need to move to next block
725
             if (offset_in_block + sizeof(entry) > block_size) {
726
               current_block = fat[current_block];
727
               offset_in_block = 0;
728
729
             continue;
730
731
732
           \ensuremath{//} format and write entry information
733
           char output_buffer[256];
734
           format_file_info(&entry, output_buffer);
735
           if (k_write(STDOUT_FILENO, output_buffer, strlen(output_buffer)) < 0) {</pre>
736
             return -1;
737
738
           // move to next entry
739
740
           offset_in_block += sizeof(entry);
          // check if we need to move to next block
if (offset_in_block + sizeof(entry) > block_size) {
741
742
743
             current_block = fat[current_block];
744
             offset_in_block = 0;
745
746
      } else {
747
        // find and display specific file
int file_offset = find_file(filename, &entry);
748
749
750
         if (file_offset < 0) {</pre>
751
         P_ERRNO = P_ENOENT;
752
          return -1;
753
754
755
        char output_buffer[256];
756
        format_file_info(&entry, output_buffer);
757
        if (k_write(STDOUT_FILENO, output_buffer, strlen(output_buffer)) < 0) {</pre>
758
          return -1;
759
      }
760
761
762
      return 0;
763 }
```

4.5.1.4 k lseek()

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

Repositions the file offset of an open file.

```
P_ERRNO = P_EINVAL;
625
626
        return -1;
627
628
      // validate the file descriptor
if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
62.9
630
631
      P_ERRNO = P_EBADF;
632
        return -1;
633
634
      \ensuremath{//} calculate new position based on whence
635
636
      int32_t new_position;
637
638
      switch (whence) {
639
       case SEEK_SET:
        new_position = offset;
640
641
          break;
        case SEEK_CUR:
642
643
         new_position = fd_table[fd].position + offset;
644
          break;
645
        case SEEK_END:
646
        new_position = fd_table[fd].size + offset;
647
          break;
648
        default:
        P_ERRNO = P_EINVAL;
649
650
          return -1;
651
652
      \ensuremath{//} check if new position is valid
653
      if (new_position < 0) {
   P_ERRNO = P_EINVAL;</pre>
654
655
656
        return -1;
657
658
659
      // update file position
     fd_table[fd].position = new_position;
660
661
662
      return new_position;
663 }
```

4.5.1.5 k_open()

Kernel-level call to open a file.

Opens a file with the specified mode.

Definition at line 33 of file fs_kfuncs.c.

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

```
58
     dir_entry_t entry;
     int file_offset = find_file(fname, &entry);
60
61
     // file exists
     if (file_offset >= 0) {
62
       // check if the file is already open in write mode by another descriptor
63
        if ((mode & (F_WRITE | F_APPEND)) != 0) {
          for (int i = 0; i < MAX_FDS; i++) {</pre>
65
66
            if (i != fd && fd_table[i].in_use &&
              strcmp(fd_table[i].filename, fname) == 0 &&
  (fd_table[i].mode & (F_WRITE | F_APPEND)) != 0) {
P_ERRNO = P_EBUSY; // file is already open for writing
67
68
69
70
               return -1;
71
72
          }
73
       }
74
        // fill in the file descriptor entry
75
        fd_table[fd].in_use = 1;
76
        fd_table[fd].ref_count++;
78
        strncpy(fd_table[fd].filename, fname, 31);
79
        fd_table[fd].filename[31] = ' \setminus 0';
        fd_table[fd].size = entry.size;
fd_table[fd].first_block = entry.firstBlock;
fd_table[fd].mode = mode;
80
81
82
83
84
        // set the initial position
85
        if (mode & F_APPEND) {
86
          fd_table[fd].position = entry.size;
87
        } else {
88
          fd_table[fd].position = 0;
89
       }
90
91
        // if mode includes {\tt F\_WRITE} and not {\tt F\_APPEND} , truncate the file
        if ((mode & F_WRITE) && !(mode & F_APPEND)) {
   // free all blocks except the first one
92
93
          uint16_t block = entry.firstBlock;
94
95
          uint16_t next_block;
          if (block != 0 && block != FAT_EOF) {
97
             next_block = fat[block];
fat[block] = FAT_EOF; // terminate the chain at the first block
98
99
100
             block = next block;
101
102
              \ensuremath{//} free the rest of the chain
103
              while (block != 0 && block != FAT_EOF) {
               next_block = fat[block];
fat[block] = FAT_FREE;
104
105
106
                block = next_block;
107
108
           }
109
110
           // update file size to 0
111
           fd_table[fd].size = 0;
           entry.size = 0;
112
           entry.mtime = time(NULL);
113
114
115
           // update the file system with the truncated file
116
           if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
117
             P_ERRNO = P_ELSEEK;
             return -1;
118
119
120
           if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
121
             P_ERRNO = P_EWRITE;
122
              return -1;
123
           }
124
       } else {
125
126
         // file doesn't exist
127
128
         // we can only create it if we are reading the file
129
         if (!(mode & F_WRITE)) {
          P_ERRNO = P_ENOENT;
130
           return -1;
131
132
133
134
         // allocate the first block
135
         uint16_t first_block = allocate_block();
         if (first_block == 0) {
  P_ERRNO = P_EFULL;
  return -1;
136
137
138
139
140
141
         // create a new file entry
142
         if (add_file_entry(fname, 0, first_block, TYPE_REGULAR, PERM_READ_WRITE) ==
143
              -1) {
144
           // error code already set by add_file_entry
```

```
145
         fat[first_block] = FAT_FREE;
146
         return -1;
147
148
149
        // fill in the file descriptor entry
150
        fd_table[fd].in_use = 1;
        fd_table[fd].ref_count++;
151
152
        strncpy(fd_table[fd].filename, fname, 31);
153
        fd_table[fd].filename[31] = ' \setminus 0';
154
        fd_table[fd].size = 0;
        fd_table[fd].first_block = first_block;
155
        fd_table[fd].position = 0;
156
157
        fd_table[fd].mode = mode;
158
159
160
     return fd;
161 }
```

4.5.1.6 k read()

Kernel-level call to read a file.

Reads data from an open file.

Definition at line 166 of file fs_kfuncs.c.

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

```
211
           // unexpected end of chain
212
          P_ERRNO = P_EINVAL;
213
          return -1;
214
215
        current_block = fat[current_block];
216
217
218
      // now we're at the right block, start reading
219
      uint32_t bytes_read = 0;
220
221
      while (bytes_read < bytes_to_read) {</pre>
        // how much data can we read from the current block
222
        uint32_t bytes_left_in_block = block_size - block_offset;
223
224
        uint32_t bytes_to_read_now =
225
            (bytes_to_read - bytes_read) < bytes_left_in_block
226
                 ? (bytes_to_read - bytes_read)
227
                 : bytes_left_in_block;
228
229
        // seek to the right position in the file
230
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + block_offset,
231
                  SEEK\_SET) == -1) {
232
          P_ERRNO = P_ELSEEK;
233
          if (bytes_read > 0) {
            fd_table[fd].position += bytes_read;
2.34
235
            return bytes_read;
236
          return -1;
237
238
239
        \ensuremath{//} read the data from the file
240
241
        ssize_t read_result = read(fs_fd, buf + bytes_read, bytes_to_read_now);
242
        if (read_result <= 0) {</pre>
243
          P_ERRNO = P_EREAD;
244
          // if we already read some data, return that count
          if (bytes_read > 0) {
  fd_table[fd].position += bytes_read;
245
246
247
            return bytes_read;
248
249
          return -1;
250
251
2.52
        bytes_read += read_result;
253
        block offset += read result;
254
255
         // if we've read all data from this block and still have more to read, go to
256
        // the next block
257
        if (block_offset == block_size && bytes_read < bytes_to_read) {</pre>
2.58
          if (current_block == FAT_EOF) {
            ^{\prime\prime} unexpected end of chain
259
260
            break:
261
262
          current_block = fat[current_block];
263
          block_offset = 0;
2.64
265
        \ensuremath{//} if we read less than expected, we might have hit EOF
266
267
        if (read_result < bytes_to_read_now) {</pre>
268
          break;
269
270
271
      // update file position
272
273
      fd_table[fd].position += bytes_read;
274
275
      return bytes_read;
276 }
```

4.5.1.7 k_unlink()

Kernel-level call to remove a file.

Removes a file from the file system.

Definition at line 566 of file fs_kfuncs.c.

```
566
       if (fname == NULL || *fname == '\0') {
  P_ERRNO = P_EINVAL;
567
568
569
         return -1;
570
571
572
       if (!is_mounted) {
573
        P_ERRNO = P_EFS_NOT_MOUNTED;
574
         return -1;
575
576
       // check if file is currently open by any process
for (int i = 0; i < MAX_FDS; i++) {
   if (fd_table[i].in_use && strcmp(fd_table[i].filename, fname) == 0) {</pre>
577
578
579
580
          P_ERRNO = P_EBUSY;
            return -1;
581
         }
582
      }
583
584
585
       // find the file in directory
       dir_entry_t entry;
int file_offset = find_file(fname, &entry);
if (file_offset < 0) {</pre>
586
587
588
        P_ERRNO = P_ENOENT;
589
590
         return -1;
591
592
593
       \ensuremath{//} mark the directory entry as deleted (set first byte to 1)
594
       entry.name[0] = 1;
595
596
       // write the modified directory entry back
       if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
597
598
599
         return -1;
600
       if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
601
       P_ERRNO = P_EWRITE;
602
         return -1;
603
604
605
606
       \ensuremath{//} free all blocks in the file chain
607
       uint16_t current_block = entry.firstBlock;
608
       uint16_t next_block;
609
       while (current_block != FAT_FREE && current_block != FAT_EOF) {
  next_block = fat[current_block];
610
611
612
         fat[current_block] = FAT_FREE;
613
         current_block = next_block;
      }
614
615
616
      return 0;
617 }
```

4.5.1.8 k_write()

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

Kernel-level call to write to a file.

Writes data to an open file.

Definition at line 281 of file fs_kfuncs.c.

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

```
291
      if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
292
       P_ERRNO = P_EBADF;
293
        return -1;
294
295
      if (str == NULL || n < 0) {</pre>
      P_ERRNO = P_EINVAL;
return -1;
296
297
298
299
      if (n == 0) {
      '" == 0)
  return 0;
}
300
301
302
303
      // check if filesystem is mounted and FAT is valid
304
      if (!is_mounted || fat == NULL) {
305
       P_ERRNO = P_EFS_NOT_MOUNTED;
306
        return -1;
307
308
309
      // get file information
310
      uint16_t current_block = fd_table[fd].first_block;
311
      uint32_t current_position = fd_table[fd].position;
312
313
      // create a local buffer for block data
      char* block_buffer = (char*)malloc(block_size);
if (block_buffer == NULL) {
314
315
      P_ERRNO = P_EMALLOC;
316
        return -1;
317
318
319
      // calculate initial block position
320
      uint32_t block_index = current_position / block_size;
321
322
      uint32_t block_offset = current_position % block_size;
323
324
      // if the file doesn't have a first block yet, allocate one
325
      if (current_block == 0) {
        current_block = allocate_block();
326
        if (current_block == 0) {
  P_ERRNO = P_EFULL;
327
328
329
          free(block_buffer);
330
          return -1;
331
332
        fd_table[fd].first_block = current_block;
333
334
335
      // navigate to the appropriate block
336
      uint16_t prev_block = 0;
337
      for (uint32_t i = 0; i < block_index; i++) {</pre>
        if (current_block == 0 || current_block == FAT_EOF ||
338
            current_block >= fat_size / 2) {
339
           // reached the end of chain prematurely, need to allocate a new block
340
          uint16_t new_block = allocate_block();
341
342
          if (new_block == 0) {
343
            P ERRNO = P EFULL;
344
            free(block_buffer);
345
            return -1;
          }
346
347
348
          // update the chain
349
          if (prev_block != 0 && prev_block < fat_size / 2) {</pre>
350
            fat[prev_block] = new_block;
351
          } else {
            // if there's no previous block, this must be the first one
352
353
            fd_table[fd].first_block = new_block;
354
355
356
          current_block = new_block;
357
358
359
        prev block = current block;
360
361
         // validate the block number before accessing FAT
362
         if (current_block >= fat_size / 2) {
          P_ERRNO = P_EINVAL;
free(block_buffer);
363
364
365
          return -1;
366
367
368
        current_block = fat[current_block];
369
370
371
      // if we ended up without a valid block, go back to the last valid one
372
      if (current_block == 0 || current_block == FAT_EOF ||
373
          current_block >= fat_size / 2) {
374
         if (prev_block != 0 && prev_block < fat_size / 2) {</pre>
375
          uint16_t new_block = allocate_block();
          if (new_block == 0) {
  P_ERRNO = P_EFULL;
376
377
```

```
free(block_buffer);
379
            return -1;
380
381
          fat[prev_block] = new_block;
382
383
          current_block = new_block;
384
        } else {
385
          P_ERRNO = P_EINVAL;
386
          free(block_buffer);
387
          return -1;
       }
388
389
390
391
      // start writing data
392
      uint32_t bytes_written = 0;
393
394
      while (bytes_written < n) {</pre>
395
        // validate current block
        if (current_block == 0 || current_block == FAT_EOF ||
396
397
            current_block >= fat_size / 2) {
398
          P_ERRNO = P_EINVAL;
399
          break;
400
       }
401
402
        \ensuremath{//} how much can we write to this block
        uint32_t space_in_block = block_size - block_offset;
403
404
        uint32_t bytes_to_write = (n - bytes_written) < space_in_block</pre>
405
                                        ? (n - bytes_written)
406
                                        : space_in_block;
407
        // position in filesystem
408
409
        off_t block_position = fat_size + (current_block - 1) * block_size;
410
411
        // if we're not writing a full block or not starting at the beginning, we
412
        // need to read-modify-write
        if (bytes_to_write < block_size || block_offset > 0) {
413
          // read the current block
if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
414
415
416
            P_ERRNO = P_ELSEEK;
417
            break;
418
419
          // read the current block data
420
421
          ssize_t read_result = read(fs_fd, block_buffer, block_size);
          if (read_result < 0) {</pre>
422
423
            P_ERRNO = P_EREAD;
424
           break;
425
426
427
          // copy the new data into the block buffer
428
          memcpy(block_buffer + block_offset, str + bytes_written, bytes_to_write);
429
430
          // seek back to write the modified block
          if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
431
432
433
            break;
434
435
436
          // write the full block back
437
          ssize_t write_result = write(fs_fd, block_buffer, block_size);
          if (write_result != block_size) {
   P_ERRNO = P_EWRITE;
438
439
440
             // we might have a partial write, but that's hard to handle correctly
441
            break;
442
443
        } else {
444
          // we're writing a full block from the beginning
          if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
P_ERRNO = P_ELSEEK;
445
446
447
            break;
448
449
450
          ssize_t write_result = write(fs_fd, str + bytes_written, bytes_to_write);
          if (write_result != bytes_to_write) {
451
            P_ERRNO = P_EWRITE;
452
453
            break;
454
455
456
        // update counters
457
        bytes_written += bytes_to_write;
458
        block_offset = (block_offset + bytes_to_write) % block_size;
459
460
461
        // if we've filled this block and still have more to write, go to the next
462
        // block
        if (block_offset == 0 && bytes_written < n) {</pre>
463
464
          // validate current block before accessing FAT
```

```
465
           if (current_block >= fat_size / 2) {
466
            P_ERRNO = P_EINVAL;
467
             break;
          }
468
469
          // check if there's a next block
470
471
          if (fat[current_block] == FAT_EOF) {
472
            // allocate a new block
473
            uint16_t new_block = allocate_block();
            if (new_block == 0) {
  P_ERRNO = P_EFULL;
474
475
476
               break;
            }
478
479
            // Update the FAT safely
480
            if (current_block < fat_size / 2) {</pre>
481
               fat[current_block] = new_block;
482
            } else {
             P_ERRNO = P_EINVAL;
483
484
               break;
485
486
487
            current_block = new_block;
488
          } else {
489
             current_block = fat[current_block];
490
491
      }
492
493
      // free the block buffer
494
495
      free (block buffer);
496
497
      // update file position
498
      fd_table[fd].position += bytes_written;
499
      // update file size if needed
500
      if (fd_table[fd].position > fd_table[fd].size) {
  fd_table[fd].size = fd_table[fd].position;
501
502
503
504
         // update the directory entry
        if dir_entry_t entry;
int dir_offset = find_file(fd_table[fd].filename, &entry);
if (dir_offset >= 0) {
  entry.size = fd_table[fd].size;
505
506
507
508
509
          entry.mtime = time(NULL);
510
511
          if (lseek(fs_fd, dir_offset, SEEK_SET) == -1) {
512
            P_ERRNO = P_ELSEEK;
            return -1;
513
514
          if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
515
516
            P_ERRNO = P_EWRITE;
517
             return -1;
518
519
      }
520
521
522
      return bytes_written;
```

4.5.2 Variable Documentation

4.5.2.1 current_fg_pid

```
pid_t current_fg_pid [extern]
```

Definition at line 31 of file kern_sys_calls.c.

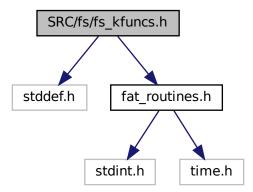
4.5.2.2 current_running_pcb

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

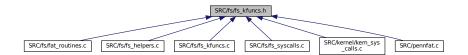
Definition at line 38 of file scheduler.c.

4.6 SRC/fs/fs_kfuncs.h File Reference

```
#include <stddef.h>
#include "fat_routines.h"
Include dependency graph for fs_kfuncs.h:
```



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



Macros

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

Functions

• int k open (const char *fname, int mode)

Opens a file with the specified mode.

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

Reads data from an open file.

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

Writes data to an open file.

• int k_close (int fd)

Closes an open file.

• int k_unlink (const char *fname)

Removes a file from the file system.

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

Repositions the file offset of an open file.

int k_ls (const char *filename)

Lists files or file information.

4.6.1 Macro Definition Documentation

4.6.1.1 SEEK_CUR

```
#define SEEK_CUR 1
```

Definition at line 17 of file fs_kfuncs.h.

4.6.1.2 SEEK_END

```
#define SEEK_END 2
```

Definition at line 18 of file fs_kfuncs.h.

4.6.1.3 SEEK_SET

```
#define SEEK_SET 0
```

Definition at line 16 of file fs_kfuncs.h.

4.6.2 Function Documentation

4.6.2.1 k_close()

```
int k\_close ( int fd)
```

Closes an open file.

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

Parameters

fd File descriptor of the open file.

Returns

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

· P_EBADF: Invalid file descriptor.

Closes an open file.

Definition at line 528 of file fs_kfuncs.c.

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

4.6.2.2 k_ls()

```
int k_ls ( \label{eq:const_char} \mbox{const_char} \ * \ \mbox{\it filename} \ )
```

Lists files or file information.

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

Parameters

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

Returns

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

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

Lists files or file information.

```
Definition at line 689 of file fs kfuncs.c.
```

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

4.6.2.3 k_lseek()

Repositions the file offset of an open file.

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

Parameters

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

Returns

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

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

Repositions the file offset of an open file.

Definition at line 622 of file fs kfuncs.c.

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

```
655    P_ERRNO = P_EINVAL;
656    return -1;
657  }
658
659    // update file position
660    fd_table[fd].position = new_position;
661
662    return new_position;
663 }
```

4.6.2.4 k_open()

Opens a file with the specified mode.

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

Parameters

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

Returns

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

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

Opens a file with the specified mode.

Definition at line 33 of file fs_kfuncs.c.

```
// validate arguments
    if (fname == NULL || *fname == '\0') {
  P_ERRNO = P_EINVAL;
35
36
       return -1:
37
38
    if ((mode & (F_READ | F_WRITE | F_APPEND)) == 0) {
39
40
       P_ERRNO = P_EINVAL;
41
       return -1;
42
43
    // check if the file system is mounted
44
    if (!is_mounted) {
45
      P_ERRNO = P_EFS_NOT_MOUNTED;
47
       return -1;
48
49
    // get a free file descriptor
50
    int fd = get_free_fd(fd_table);
    if (fd < 0) {
      P_ERRNO = P_EFULL; // no free file descriptors
      return -1;
54
5.5
56
57
    // check if the file exists
    dir_entry_t entry;
```

```
int file_offset = find_file(fname, &entry);
61
      // file exists
62
      if (file_offset >= 0) {
       // check if the file is already open in write mode by another descriptor
if ((mode & (F_WRITE | F_APPEND)) != 0) {
  for (int i = 0; i < MAX_FDS; i++) {</pre>
6.3
64
65
            if (i != fd && fd_table[i].in_use &&
67
                strcmp(fd_table[i].filename, fname) == 0 &&
              (fd_table[i].mode & (F_WRITE | F_APPEND)) != 0) {
P_ERRNO = P_EBUSY; // file is already open for writing
68
69
              return -1;
70
            }
71
72
73
74
        // fill in the file descriptor entry
75
        fd_table[fd].in_use = 1;
76
        fd_table[fd].ref_count++;
        strncpy(fd_table[fd].filename, fname, 31);
78
79
        fd_table[fd].filename[31] = ' \setminus 0';
80
        fd_table[fd].size = entry.size;
        fd_table[fd].first_block = entry.firstBlock;
81
        fd_table[fd].mode = mode;
82
83
        // set the initial position
85
        if (mode & F_APPEND) {
86
         fd_table[fd].position = entry.size;
87
         fd_table[fd].position = 0;
88
89
90
        // if mode includes F_WRITE and not F_APPEND, truncate the file
91
92
        if ((mode & F_WRITE) && !(mode & F_APPEND)) {
93
          \ensuremath{//} free all blocks except the first one
         uint16_t block = entry.firstBlock;
uint16_t next_block;
94
95
96
          if (block != 0 && block != FAT_EOF) {
            next_block = fat[block];
fat[block] = FAT_EOF; // terminate the chain at the first block
98
99
100
             block = next_block;
101
102
             // free the rest of the chain
             while (block != 0 && block != FAT_EOF) {
103
104
                next_block = fat[block];
105
                fat[block] = FAT_FREE;
106
               block = next_block;
107
             }
108
           }
109
110
           // update file size to 0
111
           fd_table[fd].size = 0;
           entry.size = 0;
entry.mtime = time(NULL);
112
113
114
115
           // update the file system with the truncated file
116
           if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
117
            P_ERRNO = P_ELSEEK;
118
             return -1;
119
120
           if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
121
             P_ERRNO = P_EWRITE;
122
             return -1;
123
124
125
       } else {
         // file doesn't exist
126
127
128
         // we can only create it if we are reading the file
         if (!(mode & F_WRITE)) {
  P_ERRNO = P_ENOENT;
129
130
           return -1;
131
132
133
134
         // allocate the first block
135
         uint16_t first_block = allocate_block();
136
         if (first_block == 0) {
          P_ERRNO = P_EFULL;
return -1;
137
138
139
140
         // create a new file entry
141
         if (add_file_entry(fname, 0, first_block, TYPE_REGULAR, PERM_READ_WRITE) ==
142
143
             -1) {
            // error code already set by add_file_entry
144
145
           fat[first_block] = FAT_FREE;
```

```
146
          return -1;
147
148
        // fill in the file descriptor entry
149
        fd_table[fd].in_use = 1;
fd_table[fd].ref_count++;
150
151
        strncpy(fd_table[fd].filename, fname, 31);
152
153
         fd_table[fd].filename[31] = ' \setminus 0';
154
        fd_table[fd].size = 0;
155
        fd_table[fd].first_block = first_block;
156
        fd_table[fd].position = 0;
157
        fd_table[fd].mode = mode;
158
159
160
      return fd;
161 }
```

4.6.2.5 k_read()

Reads data from an open file.

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

Parameters

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

Returns

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

- P_EBADF: Invalid file descriptor.
- · P EINVAL: Invalid buffer or count.

Reads data from an open file.

Definition at line 166 of file fs_kfuncs.c.

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

```
if (buf == NULL || n < 0) {</pre>
184
185
      P_ERRNO = P_EINVAL;
186
        return -1;
187
188
      if (n == 0) {
189
       return 0;
190
191
192
      // check if we're at EOF already
193
      if (fd_table[fd].position >= fd_table[fd].size) {
194
       return 0:
195
196
197
      // determine how many bytes we can actually read
198
      uint32_t bytes_to_read = n;
199
      if (fd_table[fd].position + bytes_to_read > fd_table[fd].size) {
200
        bytes_to_read = fd_table[fd].size - fd_table[fd].position;
201
202
203
      // find the block containing the current position
204
      uint16_t current_block = fd_table[fd].first_block;
      uint32_t block_index = fd_table[fd].position / block_size;
205
      uint32_t block_offset = fd_table[fd].position % block_size;
206
2.07
208
      // navigate to the correct block in the chain
      for (uint32_t i = 0; i < block_index; i++) {</pre>
209
210
           (current_block == 0 || current_block == FAT_EOF) {
211
          // unexpected end of chain
212
          P_ERRNO = P_EINVAL;
213
          return -1:
214
215
        current_block = fat[current_block];
216
217
218
      // now we're at the right block, start reading
219
      uint32_t bytes_read = 0;
220
221
      while (bytes_read < bytes_to_read) {</pre>
222
        // how much data can we read from the current block
223
        uint32_t bytes_left_in_block = block_size - block_offset;
224
        uint32_t bytes_to_read_now =
            (bytes_to_read - bytes_read) < bytes_left_in_block
     ? (bytes_to_read - bytes_read)</pre>
225
226
227
                 : bytes_left_in_block;
228
229
        \ensuremath{//} seek to the right position in the file
230
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + block_offset,
                  SEEK\_SET) == -1) {
2.31
          P_ERRNO = P_ELSEEK;
232
233
          if (bytes read > 0) {
            fd_table[fd].position += bytes_read;
234
235
            return bytes_read;
236
237
          return -1;
238
239
240
        // read the data from the file
241
        ssize_t read_result = read(fs_fd, buf + bytes_read, bytes_to_read_now);
242
        if (read_result <= 0) {</pre>
243
          P_ERRNO = P_EREAD;
          // if we already read some data, return that count
if (bytes_read > 0) {
2.44
245
246
            fd_table[fd].position += bytes_read;
247
            return bytes_read;
248
249
          return -1;
250
251
252
        bytes_read += read_result;
        block_offset += read_result;
253
254
255
        // if we've read all data from this block and still have more to read, go to
256
        // the next block
        if (block_offset == block_size && bytes_read < bytes_to_read) {</pre>
257
          if (current_block == FAT_EOF) {
258
259
            // unexpected end of chain
260
261
262
          current_block = fat[current_block];
263
          block_offset = 0;
264
265
        // if we read less than expected, we might have hit EOF
266
267
        if (read_result < bytes_to_read_now) {</pre>
268
          break;
269
270
     }
```

```
271
272  // update file position
273  fd_table[fd].position += bytes_read;
274
275  return bytes_read;
276 }
```

4.6.2.6 k unlink()

```
int k_unlink ( const\ char\ *\ \textit{fname}\ )
```

Removes a file from the file system.

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

Parameters

fname The name of the file to remove.

Returns

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

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

Removes a file from the file system.

```
Definition at line 566 of file fs kfuncs.c.
```

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

```
601
      if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
602
      P_ERRNO = P_EWRITE;
603
        return -1;
604
605
      // free all blocks in the file chain
606
607
     uint16_t current_block = entry.firstBlock;
608
     uint16_t next_block;
609
      while (current_block != FAT_FREE && current_block != FAT_EOF) {
610
      next_block = fat[current_block];
fat[current_block] = FAT_FREE;
611
612
       current_block = next_block;
613
614
615
616 return 0;
617 }
```

4.6.2.7 k write()

Writes data to an open file.

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

Parameters

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

Returns

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

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

Writes data to an open file.

Definition at line 281 of file fs kfuncs.c.

```
281
282
       // handle standard output and error
283
      if (fd == STDOUT_FILENO) {
284
        return write(STDOUT_FILENO, str, n);
285
286
      if (fd == STDERR FILENO) {
        return write(STDERR_FILENO, str, n);
287
288
289
290
      // validate inputs
      if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
  P_ERRNO = P_EBADF;
291
292
293
        return -1;
294
      if (str == NULL || n < 0) {</pre>
```

```
296
        P_ERRNO = P_EINVAL;
297
        return -1;
298
299
      if (n == 0) {
300
        return 0;
301
302
303
      \ensuremath{//} check if filesystem is mounted and FAT is valid
304
      if (!is_mounted || fat == NULL) {
305
        P_ERRNO = P_EFS_NOT_MOUNTED;
306
        return -1;
307
308
      // get file information
309
310
      uint16_t current_block = fd_table[fd].first_block;
311
      uint32_t current_position = fd_table[fd].position;
312
313
       // create a local buffer for block data
      char* block_buffer = (char*)malloc(block_size);
314
      if (block_buffer == NULL) {
315
316
       P_ERRNO = P_EMALLOC;
317
        return -1;
318
319
      // calculate initial block position
320
      uint32_t block_index = current_position / block_size;
321
322
      uint32_t block_offset = current_position % block_size;
323
324
      // if the file doesn't have a first block yet, allocate one
      if (current_block == 0) {
325
326
        current_block = allocate_block();
327
        if (current_block == 0) {
328
          P_ERRNO = P_EFULL;
329
           free(block_buffer);
330
          return -1;
331
332
        fd_table[fd].first_block = current_block;
333
334
335
      // navigate to the appropriate block
336
      uint16_t prev_block = 0;
      for (uint32_t i = 0; i < block_index; i++) {
   if (current_block == 0 || current_block == FAT_EOF ||</pre>
337
338
             current_block >= fat_size / 2) {
339
           // reached the end of chain prematurely, need to allocate a new block
340
341
           uint16_t new_block = allocate_block();
342
           if (new_block == 0) {
            P ERRNO = P EFULL:
343
             free(block_buffer);
344
             return -1;
345
346
          }
347
348
           // update the chain
349
           if (prev_block != 0 && prev_block < fat_size / 2) {</pre>
350
             fat[prev_block] = new_block;
351
           } else {
            // if there's no previous block, this must be the first one
352
353
             fd_table[fd].first_block = new_block;
354
355
356
          current_block = new_block;
357
358
        prev_block = current_block;
359
360
361
         // validate the block number before accessing FAT
362
         if (current_block >= fat_size / 2) {
          P_ERRNO = P_EINVAL;
363
          free (block_buffer);
364
365
           return -1;
366
367
368
        current_block = fat[current_block];
369
370
371
       // if we ended up without a valid block, go back to the last valid one
372
       if (current_block == 0 || current_block == FAT_EOF ||
        current_block >= fat_size / 2) {
if (prev_block != 0 && prev_block < fat_size / 2) {
  uint16_t new_block = allocate_block();</pre>
373
374
375
376
           if (new_block == 0) {
  P_ERRNO = P_EFULL;
377
378
             free (block_buffer);
379
             return -1;
380
           }
381
382
           fat[prev block] = new block;
```

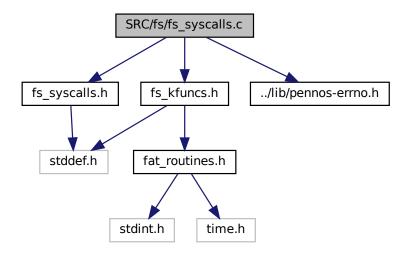
```
current_block = new_block;
384
          P_ERRNO = P_EINVAL;
385
386
          free(block_buffer);
387
          return -1;
388
        }
389
390
391
      // start writing data
392
      uint32_t bytes_written = 0;
393
394
      while (bytes_written < n) {</pre>
395
        // validate current block
396
        if (current_block == 0 || current_block == FAT_EOF ||
397
            current_block >= fat_size / 2) {
398
          P_ERRNO = P_EINVAL;
399
          break:
        }
400
401
402
        // how much can we write to this block
403
        uint32_t space_in_block = block_size - block_offset;
404
        uint32_t bytes_to_write = (n - bytes_written) < space_in_block</pre>
                                        ? (n - bytes_written)
405
406
                                        : space_in_block;
407
        // position in filesystem
408
409
        off_t block_position = fat_size + (current_block - 1) * block_size;
410
411
        // if we're not writing a full block or not starting at the beginning, we
412
        // need to read-modify-write
413
        if (bytes_to_write < block_size || block_offset > 0) {
414
          // read the current block
415
          if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
416
            P_ERRNO = P_ELSEEK;
417
            break;
418
419
420
          // read the current block data
421
          ssize_t read_result = read(fs_fd, block_buffer, block_size);
422
          if (read_result < 0) {</pre>
423
            P_ERRNO = P_EREAD;
424
            break:
425
426
          // copy the new data into the block buffer
427
428
          memcpy(block_buffer + block_offset, str + bytes_written, bytes_to_write);
429
          // seek back to write the modified block
if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
430
431
           P_ERRNO = P_ELSEEK;
432
433
            break;
434
435
436
          \ensuremath{//} write the full block back
          ssize_t write_result = write(fs_fd, block_buffer, block_size);
437
          if (write_result != block_size) {
   P_ERRNO = P_EWRITE;
438
439
440
            // we might have a partial write, but that's hard to handle correctly
441
            break;
442
443
        } else {
          // we're writing a full block from the beginning
if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
444
445
           P_ERRNO = P_ELSEEK;
446
447
            break;
448
449
          ssize_t write_result = write(fs_fd, str + bytes_written, bytes_to_write);
450
451
          if (write_result != bytes_to_write) {
452
            P_ERRNO = P_EWRITE;
453
454
          }
455
        }
456
        // update counters
457
458
        bytes_written += bytes_to_write;
459
        block_offset = (block_offset + bytes_to_write) % block_size;
460
461
        // if we've filled this block and still have more to write, go to the next
        // block
462
        if (block_offset == 0 && bytes_written < n) {</pre>
463
464
          // validate current block before accessing FAT
465
          if (current_block >= fat_size / 2) {
466
            P_ERRNO = P_EINVAL;
467
            break;
468
469
```

```
// check if there's a next block
471
            if (fat[current_block] == FAT_EOF) {
472
              // allocate a new block
              uint16_t new_block = allocate_block();
if (new_block == 0) {
   P_ERRNO = P_EFULL;
473
474
475
               break;
476
477
478
              // Update the FAT safely
if (current_block < fat_size / 2) {</pre>
479
480
481
                fat[current_block] = new_block;
482
483
                P_ERRNO = P_EINVAL;
484
                break;
485
486
487
              current_block = new_block;
           } else {
488
489
              current_block = fat[current_block];
490
491
492
      }
493
494
       // free the block buffer
495
       free(block_buffer);
496
497
       // update file position
498
       fd_table[fd].position += bytes_written;
499
500
       // update file size if needed
       fig the fire size if the detail if (fd_table[fd].position > fd_table[fd].size) {
  fd_table[fd].size = fd_table[fd].position;
501
502
503
504
          // update the directory entry
         dir_entry_t entry;
int dir_offset = find_file(fd_table[fd].filename, &entry);
if (dir_offset >= 0) {
505
506
507
508
           entry.size = fd_table[fd].size;
509
            entry.mtime = time(NULL);
510
           if (lseek(fs_fd, dir_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
511
512
              return -1;
513
514
515
            if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
             P_ERRNO = P_EWRITE;
return -1;
516
517
518
519
520
521
       return bytes_written;
```

4.7 SRC/fs/fs_syscalls.c File Reference

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

Include dependency graph for fs_syscalls.c:



Functions

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

System call to open a file.

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

System call to read from a file.

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

System call to write to a file.

• int s_close (int fd)

System call to close a file.

• int s_unlink (const char *fname)

System call to remove a file.

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

System call to reposition the file offset.

• int s_ls (const char *filename)

System call to list files.

4.7.1 Function Documentation

4.7.1.1 s_close()

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

System call to close a file.

Closes an open file descriptor.

This is a wrapper around the kernel function k_close.

Definition at line 42 of file fs_syscalls.c.

4.7.1.2 s_ls()

System call to list files.

Lists files in the current directory or displays file information.

This is a wrapper around the kernel function k_ls.

```
Definition at line 69 of file fs_syscalls.c.
```

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

4.7.1.3 s_lseek()

System call to reposition the file offset.

Repositions the file offset of an open file.

This is a wrapper around the kernel function k_lseek.

```
Definition at line 60 of file fs_syscalls.c.
```

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

4.7.1.4 s_open()

System call to open a file.

Opens a file with the specified access mode.

This is a wrapper around the kernel function k_open.

Definition at line 15 of file fs_syscalls.c.

4.7.1.5 s_read()

System call to read from a file.

Reads data from an open file.

This is a wrapper around the kernel function k_read.

```
Definition at line 24 of file fs syscalls.c.
```

4.7.1.6 s_unlink()

System call to remove a file.

Removes a file from the file system.

This is a wrapper around the kernel function k_unlink.

```
Definition at line 51 of file fs_syscalls.c.
```

4.7.1.7 s_write()

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

System call to write to a file.

Writes data to an open file.

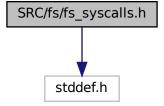
This is a wrapper around the kernel function k_write.

```
Definition at line 33 of file fs_syscalls.c.
```

4.8 SRC/fs/fs_syscalls.h File Reference

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

Include dependency graph for fs_syscalls.h:



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



Macros

- #define STDIN FILENO 0
- #define STDOUT_FILENO 1
- #define STDERR_FILENO 2

Functions

• int s open (const char *fname, int mode)

Opens a file with the specified access mode.

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

Reads data from an open file.

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

Writes data to an open file.

• int s_close (int fd)

Closes an open file descriptor.

• int s_unlink (const char *fname)

Removes a file from the file system.

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

Repositions the file offset of an open file.

• int s_ls (const char *filename)

Lists files in the current directory or displays file information.

4.8.1 Macro Definition Documentation

4.8.1.1 STDERR_FILENO

```
#define STDERR_FILENO 2
```

Definition at line 18 of file fs_syscalls.h.

4.8.1.2 STDIN_FILENO

```
#define STDIN_FILENO 0
```

Definition at line 16 of file fs_syscalls.h.

4.8.1.3 STDOUT_FILENO

```
#define STDOUT_FILENO 1
```

Definition at line 17 of file fs_syscalls.h.

4.8.2 Function Documentation

4.8.2.1 s_close()

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

Closes an open file descriptor.

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

Parameters

```
fd The file descriptor to close.
```

Returns

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

• P_EBADF: fd is not a valid file descriptor.

Closes an open file descriptor.

This is a wrapper around the kernel function k_close.

Definition at line 42 of file fs_syscalls.c.

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

4.8.2.2 s ls()

Lists files in the current directory or displays file information.

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

Parameters

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

Returns

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

• P_ENOENT: The specified file does not exist.

Lists files in the current directory or displays file information.

This is a wrapper around the kernel function k_ls.

Definition at line 69 of file fs_syscalls.c.

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

4.8.2.3 s_lseek()

Repositions the file offset of an open file.

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

Parameters

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

Returns

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

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

Repositions the file offset of an open file.

This is a wrapper around the kernel function k_lseek.

```
Definition at line 60 of file fs_syscalls.c.
```

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

4.8.2.4 s_open()

Opens a file with the specified access mode.

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

Parameters

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

Returns

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

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

Opens a file with the specified access mode.

This is a wrapper around the kernel function k_open.

```
Definition at line 15 of file fs_syscalls.c.
```

4.8.2.5 s_read()

Reads data from an open file.

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

Parameters

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

Returns

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

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

Reads data from an open file.

This is a wrapper around the kernel function k_read.

Definition at line 24 of file fs_syscalls.c.

4.8.2.6 s_unlink()

Removes a file from the file system.

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

Parameters

fname The name of the file to remov	e.
-------------------------------------	----

Returns

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

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

Removes a file from the file system.

This is a wrapper around the kernel function k_unlink.

Definition at line 51 of file fs_syscalls.c.

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

4.8.2.7 s_write()

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

Writes data to an open file.

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

Parameters

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

Returns

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

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

Writes data to an open file.

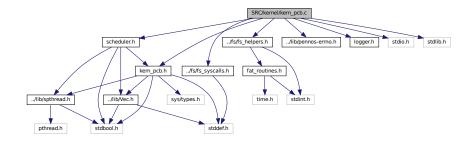
This is a wrapper around the kernel function k_write.

```
Definition at line 33 of file fs_syscalls.c.
```

4.9 SRC/kernel/kern_pcb.c File Reference

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

Include dependency graph for kern_pcb.c:



Functions

void free pcb (void *pcb)

Free resources associated with a PCB.

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

Initializes a PCB with the given parameters.

void remove_child_in_parent (pcb_t *parent, pcb_t *child)

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

pcb_t * k_proc_create (pcb_t *parent, int priority)

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

void k_proc_cleanup (pcb_t *proc)

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

Variables

- int next pid = 2
- Vec current_pcbs
- pcb_t * current_running_pcb

4.9.1 Function Documentation

4.9.1.1 create_pcb()

Initializes a PCB with the given parameters.

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

Definition at line 41 of file kern_pcb.c.

```
46
     pcb_t* ret_pcb = malloc(sizeof(pcb_t));
     if (ret_pcb == NULL) {
  perror("malloc failed for PCB creation");
47
48
       return NULL;
49
     }
50
    ret_pcb->pid = pid;
     ret_pcb->par_pid = par_pid;
     ret_pcb->priority = priority;
54
                                         // running by default
    ret_pcb->process_state = 'R';
ret_pcb->input_fd = input_fd;
5.5
56
     ret_pcb->output_fd = output_fd;
    ret_pcb->process_status = 0; // default status
50
    ret_pcb->child_pcbs = vec_new(0, NULL); // NULL deconstructor prevents
60
                                                    // double free
61
62
    for (int i = 0; i < 3; i++) {
63
       ret_pcb->signals[i] = false;
65 }
66
    ret_pcb->is_sleeping = false;
ret_pcb->time_to_wake = -1; // default to not sleeping
67
68
69
     return ret_pcb;
```

4.9.1.2 free_pcb()

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

Free resources associated with a PCB.

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

Definition at line 29 of file kern_pcb.c.

4.9.1.3 k proc cleanup()

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

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

Definition at line 150 of file kern_pcb.c.

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

```
188
      spthread_cancel (proc->thread_handle);
      spthread_continue(proc->thread_handle);
189
190
      spthread_suspend(proc->thread_handle);
191
      spthread_join(proc->thread_handle, NULL);
192
     // delete this process from any queue it's in + free it
193
     delete_process_from_all_queues(proc);
194
195
     free_pcb(proc);
196 }
```

4.9.1.4 k_proc_create()

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

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

Definition at line 93 of file kern pcb.c.

```
94
      if (parent == NULL) { // init creation case
        pcb_t* init = create_pcb(1, 0, 0, 0, 1);
if (init == NULL) {
9.5
96
          P_ERRNO = P_ENULL;
98
          return NULL;
99
100
         init->fd_table[0] = STDIN_FILENO;
         init->fd_table[1] = STDOUT_FILENO;
init->fd_table[2] = STDERR_FILENO;
101
102
103
         for (int i = 3; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {</pre>
104
           init->fd_table[i] = -1;
105
106
        increment_fd_ref_count(STDIN_FILENO);
increment_fd_ref_count(STDOUT_FILENO);
increment_fd_ref_count(STDERR_FILENO);
107
108
109
110
111
         current_running_pcb = init;
112
         put_pcb_into_correct_queue(init);
113
         vec_push_back(&current_pcbs, init);
114
         return init;
115
116
117
       pcb_t* child = create_pcb(next_pid++, parent->pid, priority, parent->input_fd,
118
                                    parent->output_fd);
       if (child == NULL) {
119
       P_ERRNO = P_ENULL;
120
121
         return NULL;
122
123
124
       // copy parent's fd table
125
       for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {</pre>
        child->fd_table[i] = parent->fd_table[i];
126
127
128
       for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {</pre>
130
        if (child->fd_table[i] != -1) {
131
           increment_fd_ref_count(child->fd_table[i]);
132
        }
133
134
135
       // update parent as needed
136
       vec_push_back(&parent->child_pcbs, child);
137
138
       // add to appropriate queue
      put_pcb_into_correct_queue(child);
vec_push_back(&current_pcbs, child);
139
140
141
142
      return child;
143 }
```

4.9.1.5 remove_child_in_parent()

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

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

Definition at line 76 of file kern_pcb.c.

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

82    }
83    }
84 }
```

4.9.2 Variable Documentation

4.9.2.1 current_pcbs

```
Vec current_pcbs [extern]
```

Definition at line 30 of file scheduler.c.

4.9.2.2 current_running_pcb

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

Definition at line 38 of file scheduler.c.

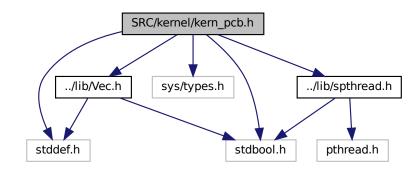
4.9.2.3 next_pid

```
int next\_pid = 2
```

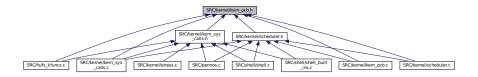
Definition at line 16 of file kern_pcb.c.

4.10 SRC/kernel/kern_pcb.h File Reference

```
#include <stddef.h>
#include <stdbool.h>
#include <sys/types.h>
#include "../lib/spthread.h"
#include "../lib/Vec.h"
Include dependency graph for kern pcb.h:
```



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



Classes

• struct pcb_st

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

Macros

• #define FILE_DESCRIPTOR_TABLE_SIZE 100

Typedefs

• typedef struct pcb_st pcb_t

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

Functions

- pcb t * create pcb (pid t pid, pid t par pid, int priority, int input fd, int output fd)
 - Creates a new PCB and initializes its fields. Notably, the thread handle and cmd are left out. It's up to the user to assign them post-call.
- void free pcb (void *pcb)

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

void remove_child_in_parent (pcb_t *parent, pcb_t *child)

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

pcb_t * k_proc_create (pcb_t *parent, int priority)

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

void k_proc_cleanup (pcb_t *proc)

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

4.10.1 Macro Definition Documentation

4.10.1.1 FILE DESCRIPTOR TABLE SIZE

```
#define FILE_DESCRIPTOR_TABLE_SIZE 100
```

Definition at line 17 of file kern_pcb.h.

4.10.2 Typedef Documentation

4.10.2.1 pcb_t

```
typedef struct pcb_st pcb_t
```

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

4.10.3 Function Documentation

4.10.3.1 create_pcb()

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

Parameters

pid	the new process id
par_pid	the parent process id
priority	the priority level (0,1,2)
input_fd	input fd
output⊷	output fd
_fd	

Returns

pointer to the newly created and malloced PCB or NULL if failure

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

Definition at line 41 of file kern_pcb.c.

```
pcb_t* ret_pcb = malloc(sizeof(pcb_t));
46
     if (ret_pcb == NULL) {
  perror("malloc failed for PCB creation");
49
       return NULL;
50
51
    ret_pcb->pid = pid;
ret_pcb->par_pid = par_pid;
ret_pcb->priority = priority;
52
53
    ret_pcb->process_state = 'R'
                                        // running by default
     ret_pcb->input_fd = input_fd;
57
    ret_pcb->output_fd = output_fd;
58
    ret_pcb->process_status = 0; // default status
59
    ret_pcb->child_pcbs = vec_new(0, NULL); // NULL deconstructor prevents
60
63
    for (int i = 0; i < 3; i++) {</pre>
64
      ret_pcb->signals[i] = false;
65
66
    ret_pcb->is_sleeping = false;
    ret_pcb->time_to_wake = -1; // default to not sleeping
68
69
70
    return ret_pcb;
```

4.10.3.2 free pcb()

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

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

Parameters

```
pcb Pointer to the PCB to be freed, NULL if error
```

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

Definition at line 29 of file kern_pcb.c.

4.10.3.3 k proc cleanup()

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

Parameters

```
proc a pcb ptr to the terminated/finished thread
```

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

Definition at line 150 of file kern_pcb.c.

```
150
       // if proc has parent (i.e. isn't init) then remove it from parent's child
152
       // list
      pcb_t* par_pcb = get_pcb_in_queue(&current_pcbs, proc->par_pid);
153
      if (par_pcb != NULL) {
154
        remove_child_in_parent (par_pcb, proc);
155
      } else {
156
      P_ERRNO = P_ENULL;
157
158
        return;
159
160
161
      // if proc has children, remove them and assign them to init parent
      if (vec_len(&proc->child_pcbs) > 0) {
   // retrieve the init process
162
163
164
        pcb_t* init_pcb =
165
             get_pcb_in_queue(&current_pcbs, 1); // init process has pid 1
166
        while (vec_len(&proc->child_pcbs) > 0) {
  pcb_t* curr_child = vec_get(&proc->child_pcbs, 0);
167
168
           vec_push_back(&init_pcb->child_pcbs, curr_child);
169
170
           vec_erase_no_deletor(&proc->child_pcbs, 0); // don't free in erase
           curr_child->par_pid = 1; // update parent to init (pid 1) log_generic_event('0', curr_child->pid, curr_child->priority,
171
172
173
                              curr child->cmd str);
174
175
      }
176
177
      // decr reference counts + close files if necessary
178
      for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {</pre>
        if (proc->fd_table[i] != -1) {
179
          if (decrement_fd_ref_count(proc->fd_table[i]) == 0) {
180
181
            s close(
182
                 proc->fd_table[i]); // close the fd since no other process using
183
184
        }
      }
185
186
      // cancel + join this thread
187
      spthread_cancel(proc->thread_handle);
189
      spthread_continue(proc->thread_handle);
190
      spthread_suspend(proc->thread_handle);
191
      spthread_join(proc->thread_handle, NULL);
192
193
      // delete this process from any queue it's in + free it
194
      delete_process_from_all_queues(proc);
```

4.10.3.4 k_proc_create()

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

Parameters

parent	a pointer to the parent pcb
priority	the priority of the child, usually 1 but exceptions like shell exist

Returns

Reference to the child PCB or NULL if error

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

Definition at line 93 of file kern pcb.c.

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

```
134
135  // update parent as needed
136  vec_push_back(&parent->child_pcbs, child);
137
138  // add to appropriate queue
139  put_pcb_into_correct_queue(child);
140  vec_push_back(&current_pcbs, child);
141
142  return child;
143 }
```

4.10.3.5 remove_child_in_parent()

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

Parameters

parent	a ptr to the parent pcb with the child list
child	a ptr to the child pcb that we'd like to remove

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

Definition at line 76 of file kern_pcb.c.

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

3  }

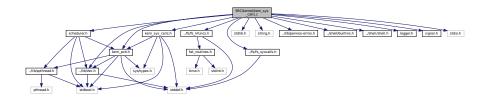
4 }
```

4.11 SRC/kernel/kern_sys_calls.c File Reference

```
#include "kern_sys_calls.h"
#include <stdlib.h>
#include <string.h>
#include "../fs/fs_kfuncs.h"
#include "../lib/Vec.h"
#include "../lib/pennos-errno.h"
#include "../shell/builtins.h"
#include "../shell/shell.h"
#include "kern_pcb.h"
#include "logger.h"
#include "scheduler.h"
#include "signal.h"
#include "../fs/fs_syscalls.h"
```

#include <stdio.h>

Include dependency graph for kern_sys_calls.c:



Functions

• int determine_index_in_queue (Vec *queue, int pid)

Determines the index of a PCB in a given queue.

void move_pcb_correct_queue (int prev_priority, int new_priority, pcb_t *curr_pcb)

Moves a PCB from its previous priority queue to its new priority queue.

void delete_from_queue (int queue_id, int pid)

Deletes a PCB from the specified queue based on its PID.

• void delete from explicit queue (Vec *queue to delete from, int pid)

Deletes a PCB from the specified explicit queue based on its PID.

void * init_func (void *input)

The function that runs the shell process.

• pid_t s_spawn_init ()

Creates the init process and spawns the shell process.

void s_cleanup_init_process ()

Cleans up Init's resources.

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

Spawns a child process with the given function and arguments.

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

Waits for a child of the calling process.

• int s_kill (pid_t pid, int signal)

Sends a signal to a process with specified pid.

void s_exit (void)

Exits the current process and cleans up its resources.

• int s_nice (pid_t pid, int priority)

Sets the priority of a process with specified pid.

void s_sleep (unsigned int ticks)

Suspends the current process for a specified number of ticks.

void * s_echo (void *arg)

System-level wrapper for the shell built-in command "echo".

void * s_ps (void *arg)

System-level wrapper for the shell built-in command "ps".

Variables

- Vec zero_priority_queue
- · Vec one priority queue
- Vec two_priority_queue
- · Vec zombie queue
- Vec sleep_blocked_queue
- Vec current pcbs
- pcb t * current running pcb
- int tick_counter
- pid_t current_fg_pid = 2

4.11.1 Function Documentation

4.11.1.1 delete_from_explicit_queue()

Deletes a PCB from the specified explicit queue based on its PID.

Helper function that deletes the given PCB from the explicit queue passed in. Notably, it does not free the PCB but instead uses vec_erase_no_deletor to remove it from the queue.

Definition at line 107 of file kern sys calls.c.

```
107

108 int index = determine_index_in_queue(queue_to_delete_from, pid);

109 if (index != -1) {

110     vec_erase_no_deletor(queue_to_delete_from, index);

111 }

112 }
```

4.11.1.2 delete from queue()

Deletes a PCB from the specified queue based on its PID.

Deletes the PCB with the specified PID from one of the priority queues, selected by the provided queue_id (0, 1, or 2).

Definition at line 88 of file kern sys calls.c.

```
89
     Vec* queue = NULL;
90
     if (queue_id == 0) {
    queue = &zero_priority_queue;
} else if (queue_id == 1) {
91
92
93
       queue = &one_priority_queue;
       queue = &two_priority_queue;
96
97
98
    int index = determine_index_in_queue(queue, pid);
if (index != -1) {
99
         vec_erase_no_deletor(queue, index);
101
102 }
```

4.11.1.3 determine_index_in_queue()

Determines the index of a PCB in a given queue.

Given a thread pid and Vec* queue, this helper function determines the vector index of the thread/pid in the queue. If the thread/pid is not found, it returns -1.

Definition at line 40 of file kern sys calls.c.

4.11.1.4 init_func()

The function that runs the shell process.

The init process function. It spawns the shell process and reaps zombie children.

Definition at line 117 of file kern_sys_calls.c.

```
118
     char* shell_argv[] = {"shell", NULL};
119
     s_spawn(shell, shell_argv, STDIN_FILENO, STDOUT_FILENO);
120
121
     // continuously wait for and reap zombie children
122
     while (true) {
     int status;
123
124
       s_waitpid(-1, &status, false);
125
127
     return NULL; // should never reach
128 }
```

4.11.1.5 move_pcb_correct_queue()

Moves a PCB from its previous priority queue to its new priority queue.

Given a thread's previous priority, this helper checks if the thread is present in that priority's queue, removes it from that queue if so, and then puts it into the new priority level's queue.

Definition at line 54 of file kern_sys_calls.c.

```
57
     Vec* prev_queue;
58
    Vec* new_queue;
59
60
    if (prev_priority == 0) {
      prev_queue = &zero_priority_queue;
61
   } else if (prev_priority == 1) {
      prev_queue = &one_priority_queue;
64
      prev_queue = &two_priority_queue;
65
66
   if (new_priority == 0) {
69
      new_queue = &zero_priority_queue;
70
   } else if (new_priority == 1) {
71
      new_queue = &one_priority_queue;
   } else {
72
73
      new_queue = &two_priority_queue;
75
76
    // delete from prev_queue, if it's present at all
77
    int ind = determine_index_in_queue(prev_queue, curr_pcb->pid);
if (ind != -1) {
78
79
      vec_erase_no_deletor(prev_queue, ind);
82
    vec_push_back(new_queue, curr_pcb);
83 1
```

4.11.1.6 s_cleanup_init_process()

```
void s_cleanup_init_process ( )
```

Cleans up Init's resources.

Wrapper system-level function to be called in pennos's main method to clean up the init process.

Definition at line 157 of file kern_sys_calls.c.

4.11.1.7 s_echo()

```
void* s_echo (
     void * arg )
```

System-level wrapper for the shell built-in command "echo".

Parameters

```
arg the pass along arguments to the u_echo function
```

Returns

NULL, dummy return value

Definition at line 358 of file kern_sys_calls.c.

```
358
359
     char** argv = (char**)arg;
     if (argv[1] == NULL) { // no args case
360
     s_exit();
361
362
       return NULL;
363
364
     365
366
      s_write(current_running_pcb->output_fd, argv[i], strlen(argv[i]));
s_write(current_running_pcb->output_fd, " ", 1);
367
368
369
370
371
372
     s_write(current_running_pcb->output_fd, "\n", 1);
373 return NULL;
374 }
```

4.11.1.8 s_exit()

```
void s_exit (
          void )
```

Exits the current process and cleans up its resources.

Unconditionally exit the calling process.

Definition at line 293 of file kern_sys_calls.c.

```
294
      // Set process state to zombie
     current_running_pcb->process_state = 'Z';
295
     current_running_pcb->process_status = 20; // EXITED_NORMALLY
296
297
298
     // Log the exit
299
     log_generic_event('E', current_running_pcb->pid,
300
                        current_running_pcb->priority,
                       current_running_pcb->cmd_str);
301
302
303
     delete_from_queue(current_running_pcb->priority, current_running_pcb->pid);
305
     log_generic_event('Z', current_running_pcb->pid,
306
                       current_running_pcb->priority,
307
                       current_running_pcb->cmd_str);
308 }
```

4.11.1.9 s_kill()

Sends a signal to a process with specified pid.

Send a signal to a particular process.

Definition at line 279 of file kern_sys_calls.c.

```
pcb_t* pcb_with_pid = get_pcb_in_queue(&current_pcbs, pid);
if (pcb_with_pid == NULL) {
    return -1; // pid not found case
}

pcb_with_pid->signals[signal] = true; // signal flagged
log_generic_event('S', pid, pcb_with_pid->priority, pcb_with_pid->cmd_str);
return 0;
```

4.11.1.10 s_nice()

Sets the priority of a process with specified pid.

Set the priority of the specified thread.

Definition at line 313 of file kern sys calls.c.

```
313
314
       if (priority < 0 || priority > 2) { // error check
315
        return -1;
316
317
      pcb_t* curr_pcb = get_pcb_in_queue(&current_pcbs, pid);
if (curr_pcb != NULL) { // found + exists
  move_pcb_correct_queue(curr_pcb->priority, priority, curr_pcb);
318
319
320
321
         log_nice_event(pid, curr_pcb->priority, priority, curr_pcb->cmd_str);
322
        curr_pcb->priority = priority;
      return 0;
323
324
325
326
      return -1; // pid not found
```

4.11.1.11 s ps()

```
void* s_ps (
     void * arg )
```

System-level wrapper for the shell built-in command "ps".

Parameters

arg the pass along arguments to the u_ps function

Returns

NULL, dummy return value

Definition at line 379 of file kern_sys_calls.c.

```
380
       char \ pid\_top[] = "PID\tPPID\tPRI\tSTAT\tCMD\n";
       s_write(current_running_pcb->output_fd, pid_top, strlen(pid_top));
for (int i = 0; i < vec_len(&current_pcbs); i++) {
   pcb_t* curr_pcb = (pcb_t*)vec_get(&current_pcbs, i);</pre>
381
382
383
384
         char buffer[100];
385
         snprintf(buffer, sizeof(buffer), "%d\t%d\t%d\t%c\t%s\n", curr_pcb->pid,
386
                    curr_pcb->par_pid, curr_pcb->priority, curr_pcb->process_state,
387
                     curr_pcb->cmd_str);
388
         s_write(current_running_pcb->output_fd, buffer, strlen(buffer));
      }
389
390
      return NULL;
391 }
```

4.11.1.12 s_sleep()

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

Suspends the current process for a specified number of ticks.

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

Definition at line 332 of file kern_sys_calls.c.

```
332
333
      if (ticks <= 0) {
334
        P_ERRNO = P_EINVAL;
335
        return;
336
337
338
     // block current process, set state to sleep
339
     current_running_pcb->process_state = 'B';
340
      current_running_pcb->is_sleeping = true;
341
      current_running_pcb->time_to_wake = tick_counter + ticks;
     log_generic_event('B', current_running_pcb->pid,
342
343
                        current_running_pcb->priority,
current_running_pcb->cmd_str);
344
345
     if (spthread_suspend(current_running_pcb->thread_handle) != 0) { // give scheduler control
346
       perror("Error in spthread_suspend in s_sleep call");
347
348 1
```

4.11.1.13 s_spawn()

Spawns a child process with the given function and arguments.

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

Definition at line 164 of file kern_sys_calls.c.

```
164
                                                                          {
165
      pcb_t* child;
        (strcmp(argv[0], "shell") == 0) {
166
167
       child = k_proc_create(current_running_pcb, 0);
168
169
       child = k_proc_create(current_running_pcb, 1);
170
171
172
      if (child == NULL) {
173
       P_ERRNO = P_ENULL;
174
       return -1;
175
176
177
     spthread t thread handle:
179
     perror("Error in spthread_create in s_spawn call");
}
     if (spthread_create(&thread_handle, NULL, func, argv) != 0) {
180
181
182
183
     child->cmd str = strdup(argv[0]);
     child->thread_handle = thread_handle;
184
185
      child->input_fd = fd0;
186
      child->output_fd = fd1;
     child->fd_table[0] = fd0;
187
188
     child->fd_table[1] = fd1;
189
190
     log_generic_event('C', child->pid, child->priority, child->cmd_str);
191
192
     return child->pid;
193 }
```

4.11.1.14 s_spawn_init()

```
pid_t s_spawn_init ( )
```

Creates the init process and spawns the shell process.

Similar to s_spawn except only called when you want to spawn the init process. It will create the init process and also spawn in the shell.

Definition at line 137 of file kern_sys_calls.c.

```
137
      pcb_t* init = k_proc_create(NULL, 0);
      if (init == NULL) {
139
      P_ERRNO = P_ENULL;
140
141
       return -1;
142
143
144
     spthread_t thread_handle;
145
      if (spthread_create(&thread_handle, NULL, init_func, NULL) != 0) {
146
       perror("Error in spthread_create in s_spawn_init call");
147
148
149
     init->cmd_str = strdup("init");
     init->thread_handle = thread_handle;
151
    return init->pid;
152 }
```

4.11.1.15 s_waitpid()

Waits for a child of the calling process.

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

Definition at line 198 of file kern_sys_calls.c.

```
198
199
       pcb_t* parent = current_running_pcb;
200
       if (parent == NULL) {
201
         return -1;
202
203
204
       // if no children, return -1
205
       bool has_child = false;
for (int i = 0; i < vec_len(&current_pcbs); i++) {</pre>
206
        pcb_t* child = vec_get(&current_pcbs, i);
207
208
         if (child->par_pid == parent->pid) {
209
          has_child = true;
210
            break;
         }
211
212
213
       if (!has_child) {
214
         return -1;
215
216
217
       \ensuremath{//} Scan the zombie queue first for terminated children.
       for (int i = 0; i < vec_len(&zombie_queue); i++) {
  pcb_t* child = vec_get(&zombie_queue, i);
  if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
218
219
220
221
          if (wstatus != NULL) {
222
              *wstatus = child->process_status;
223
            log_generic_event('W', child->pid, child->priority, child->cmd_str);
vec_erase_no_deletor(&zombie_queue, i);
224
225
226
            delete_from_explicit_queue(&parent->child_pcbs, child->pid);
            k_proc_cleanup(child);
```

```
return child->pid;
229
230
231
2.32
        \ensuremath{//} If nohang is true, return immediately if no child has exited
233
        if (nohang) {
234
         return 0;
235
236
237
        \ensuremath{//} Block the parent until a child exits
       delete_from_queue(parent->priority, parent->pid);
parent->process_state = 'B';
238
239
        log_generic_event('B', parent->pid, parent->priority, parent->cmd_str);
240
241
242
         // Scan the zombie queue first for terminated children.
for (int i = 0; i < vec_len(&zombie_queue); i++) {
  pcb_t* child = vec_get(&zombie_queue, i);</pre>
243
244
245
             if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
246
247
               if (wstatus != NULL) {
248
                  *wstatus = child->process_status;
249
               log_generic_event('W', child->pid, child->priority, child->cmd_str);
vec_erase_no_deletor(&zombie_queue, i);
delete_from_explicit_queue(&parent->child_pcbs, child->pid);
2.50
2.51
252
                k_proc_cleanup(child);
254
                return child->pid;
255
256
         }
257
          // scan children of current running process for non-terminated state changes for (int i = 0; i < vec\_len(\&parent->child\_pcbs); i++) {
258
259
260
           pcb_t* child = vec_get(&parent->child_pcbs, i);
261
              if ((pid == -1 || child->pid == pid) && (child->process_status == 21 || child->process_status ==
         23)) { // signaled
  if (wstatus != NULL) {
   *wstatus = child->process_status;
262
263
264
                log_generic_event('W', child->pid, child->priority, child->cmd_str);
child->process_status = 0; // reset status
265
266
267
                return child->pid;
2.68
             }
269
          }
270
       }
271
272
        // If we get here, something went wrong
273
       return -1;
274 }
```

4.11.2 Variable Documentation

4.11.2.1 current_fg_pid

```
pid_t current_fg_pid = 2
```

Definition at line 31 of file kern_sys_calls.c.

4.11.2.2 current_pcbs

```
Vec current pcbs [extern]
```

Definition at line 30 of file scheduler.c.

4.11.2.3 current_running_pcb

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

Definition at line 38 of file scheduler.c.

4.11.2.4 one_priority_queue

```
Vec one_priority_queue [extern]
```

Definition at line 25 of file scheduler.c.

4.11.2.5 sleep_blocked_queue

```
Vec sleep_blocked_queue [extern]
```

Definition at line 28 of file scheduler.c.

4.11.2.6 tick_counter

```
int tick_counter [extern]
```

Definition at line 35 of file scheduler.c.

4.11.2.7 two_priority_queue

```
Vec two_priority_queue [extern]
```

Definition at line 26 of file scheduler.c.

4.11.2.8 zero_priority_queue

```
Vec zero_priority_queue [extern]
```

Definition at line 24 of file scheduler.c.

4.11.2.9 zombie_queue

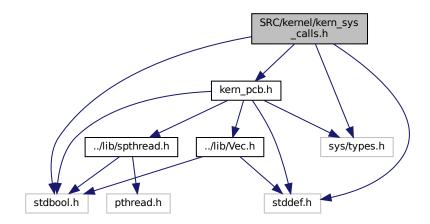
```
Vec zombie_queue [extern]
```

Definition at line 27 of file scheduler.c.

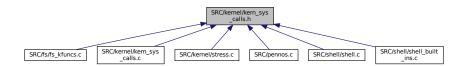
4.12 SRC/kernel/kern_sys_calls.h File Reference

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

Include dependency graph for kern_sys_calls.h:



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



Functions

• int determine_index_in_queue (Vec *queue, int pid)

Given a thread pid and Vec* queue, this helper function determines the vector index of the thread/pid in the queue. If the thread/pid is not found, it returns -1.

• void move pcb correct queue (int prev priority, int new priority, pcb t *curr pcb)

Given a thread's previous priority, this helper checks if the thread is present in that priority's queue, removes it from that queue if so, and then puts it into the new priority level's queue.

void delete_from_queue (int queue_id, int pid)

Deletes the PCB with the specified PID from one of the priority queues, selected by the provided queue_id (0, 1, or 2)

• void delete from explicit queue (Vec *queue to delete from, int pid)

Helper function that deletes the given PCB from the explicit queue passed in. Notably, it does not free the PCB but instead uses vec_erase_no_deletor to remove it from the queue.

void * init_func (void *input)

The init process function. It spawns the shell process and reaps zombie children.

• pid ts spawn init()

Similar to s_spawn except only called when you want to spawn the init process. It will create the init process and also spawn in the shell.

void s_cleanup_init_process ()

Wrapper system-level function to be called in pennos's main method to clean up the init process.

pid ts spawn (void *(*func)(void *), char *argv[], int fd0, int fd1)

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

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

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

int s kill (pid t pid, int signal)

Send a signal to a particular process.

void s_exit (void)

Unconditionally exit the calling process.

• int s_nice (pid_t pid, int priority)

Set the priority of the specified thread.

void s_sleep (unsigned int ticks)

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

void * s_echo (void *arg)

System-level wrapper for the shell built-in command "echo".

void * s_ps (void *arg)

System-level wrapper for the shell built-in command "ps".

4.12.1 Function Documentation

4.12.1.1 delete_from_explicit_queue()

Helper function that deletes the given PCB from the explicit queue passed in. Notably, it does not free the PCB but instead uses vec_erase_no_deletor to remove it from the queue.

Parameters

queue_to_delete_from	ptr to Vec* queue to delete from
pid	the pid of the PCB to delete

Helper function that deletes the given PCB from the explicit queue passed in. Notably, it does not free the PCB but instead uses vec erase no deletor to remove it from the queue.

Definition at line 107 of file kern_sys_calls.c.

```
107
108 int index = determine_index_in_queue(queue_to_delete_from, pid);
109 if (index != -1) {
110     vec_erase_no_deletor(queue_to_delete_from, index);
111    }
112 }
```

4.12.1.2 delete from queue()

Deletes the PCB with the specified PID from one of the priority queues, selected by the provided queue_id (0, 1, or 2).

Parameters

queue⊷	An integer representing the queue: 0 for zero_priority_queue, 1 for one_priority_queue, or 2 for
_id	two_priority_queue.
pid	The PID of the PCB to be removed.

Deletes the PCB with the specified PID from one of the priority queues, selected by the provided queue_id (0, 1, or 2).

Definition at line 88 of file kern_sys_calls.c.

```
Vec* queue = NULL;
    if (queue_id == 0) {
90
   queue = &zero_priority_queue;
} else if (queue_id == 1) {
93
      queue = &one_priority_queue;
   } else {
94
95
       queue = &two_priority_queue;
96
98 int index = determine_index_in_queue(queue, pid);
99
    if (index != -1) {
100
        vec_erase_no_deletor(queue, index);
      }
101
```

4.12.1.3 determine_index_in_queue()

Given a thread pid and Vec* queue, this helper function determines the vector index of the thread/pid in the queue. If the thread/pid is not found, it returns -1.

Parameters

queue	pointer to the vector queue that may contain the thread/pid
pid	the thread's pid

Returns

the index of the thread/pid in the queue, or -1 if not found

Given a thread pid and Vec* queue, this helper function determines the vector index of the thread/pid in the queue. If the thread/pid is not found, it returns -1.

Definition at line 40 of file kern_sys_calls.c.

4.12.1.4 init_func()

The init process function. It spawns the shell process and reaps zombie children.

Parameters

```
input unused but needed for typing reasons
```

Returns

irrelvant return value because never supposed to return

The init process function. It spawns the shell process and reaps zombie children.

Definition at line 117 of file kern_sys_calls.c.

```
char* shell_argv[] = {"shell", NULL};
s_spawn(shell, shell_argv, STDIN_FILENO, STDOUT_FILENO);
118
119
120
121
       \ensuremath{//} continuously wait for and reap zombie children
122
      while (true) {
       int status;
123
124
        s_waitpid(-1, &status, false);
125
126
      return NULL; // should never reach
128 }
```

4.12.1.5 move_pcb_correct_queue()

Given a thread's previous priority, this helper checks if the thread is present in that priority's queue, removes it from that queue if so, and then puts it into the new priority level's queue.

Parameters

prev_priority	thread's previous priority
new_priority	thread's new priority
curr_pcb	pointer to the thread's PCB

Precondition

assumes the prev_priority and new_priority falls in integers [0, 2]

Given a thread's previous priority, this helper checks if the thread is present in that priority's queue, removes it from that queue if so, and then puts it into the new priority level's queue.

Definition at line 54 of file kern sys calls.c.

```
Vec* prev_queue;
58
     Vec* new_queue;
59
    if (prev_priority == 0) {
60
      prev_queue = &zero_priority_queue;
61
    } else if (prev_priority == 1) {
62
      prev_queue = &one_priority_queue;
6.5
      prev_queue = &two_priority_queue;
66
67
    if (new priority == 0) {
68
      new_queue = &zero_priority_queue;
70
    } else if (new_priority == 1) {
      new_queue = &one_priority_queue;
72
    } else
73
      new_queue = &two_priority_queue;
74
75
    // delete from prev_queue, if it's present at all
    int ind = determine_index_in_queue(prev_queue, curr_pcb->pid);
78
    if (ind != -1) {
79
       vec_erase_no_deletor(prev_queue, ind);
80
81
    vec_push_back(new_queue, curr_pcb);
```

4.12.1.6 s cleanup init process()

```
void s_cleanup_init_process ( )
```

Wrapper system-level function to be called in pennos's main method to clean up the init process.

Wrapper system-level function to be called in pennos's main method to clean up the init process.

Definition at line 157 of file kern_sys_calls.c.

```
157 {
158 k_proc_cleanup(get_pcb_in_queue(&current_pcbs, 1));
159 }
```

4.12.1.7 s_echo()

```
void* s_echo (
     void * arg )
```

System-level wrapper for the shell built-in command "echo".

Parameters

arg the pass along arguments to the u_echo function

Returns

NULL, dummy return value

Definition at line 358 of file kern_sys_calls.c.

```
char** argv = (char**)arg;
if (argv[1] == NULL) { // no args case
359
360
     s_exit();
return NULL;
361
362
363
    364
365
366
367
368
369
370
371
372
    s_write(current_running_pcb->output_fd, "\n", 1);
373
    return NULL;
374 }
```

4.12.1.8 s_exit()

```
void s_exit (
          void )
```

Unconditionally exit the calling process.

Unconditionally exit the calling process.

Definition at line 293 of file kern_sys_calls.c.

```
293
294
     // Set process state to zombie
295
     current_running_pcb->process_state = 'Z';
296
     current_running_pcb->process_status = 20; // EXITED_NORMALLY
297
     // Log the exit
298
     log_generic_event('E', current_running_pcb->pid,
299
300
                       current_running_pcb->priority,
301
                       current_running_pcb->cmd_str);
302
303
     delete_from_queue(current_running_pcb->priority, current_running_pcb->pid);
304
     log_generic_event('Z', current_running_pcb->pid,
305
306
                       current_running_pcb->priority,
307
                       current_running_pcb->cmd_str);
308 }
```

4.12.1.9 s_kill()

Send a signal to a particular process.

Parameters

pid	Process ID of the target proces.
signal	Signal number to be sent 0 = P_SIGSTOP, 1 = P_SIGCONT, 2 = P_SIGTERM

Returns

0 on success, -1 on error.

Send a signal to a particular process.

Definition at line 279 of file kern_sys_calls.c.

```
279
280    pcb_t* pcb_with_pid = get_pcb_in_queue(&current_pcbs, pid);
281    if (pcb_with_pid == NULL) {
282        return -1;    // pid not found case
283    }
284
285    pcb_with_pid->signals[signal] = true;    // signal flagged
286    log_generic_event('S', pid, pcb_with_pid->priority, pcb_with_pid->cmd_str);
287    return 0;
288 }
```

4.12.1.10 s_nice()

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

Set the priority of the specified thread.

Parameters

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

Returns

0 on success, -1 on failure.

Set the priority of the specified thread.

Definition at line 313 of file kern_sys_calls.c.

```
313
       if (priority < 0 || priority > 2) { // error check
314
315
316
317
       pcb_t* curr_pcb = get_pcb_in_queue(&current_pcbs, pid);
if (curr_pcb != NULL) {    // found + exists
    move_pcb_correct_queue(curr_pcb->priority, priority, curr_pcb);
318
319
320
321
         log_nice_event(pid, curr_pcb->priority, priority, curr_pcb->cmd_str);
322
        curr_pcb->priority = priority;
323
         return 0;
324 }
325
326
      return -1; // pid not found
```

4.12.1.11 s_ps()

System-level wrapper for the shell built-in command "ps".

Parameters

```
arg the pass along arguments to the u_ps function
```

Returns

NULL, dummy return value

Definition at line 379 of file kern_sys_calls.c.

```
char pid_top[] = "PID\tPPID\tPRI\tSTAT\tCMD\n";
380
       s_write(current_running_pcb->output_fd, pid_top, strlen(pid_top));
for (int i = 0; i < vec_len(&current_pcbs); i++) {
   pcb_t* curr_pcb = (pcb_t*)vec_get(&current_pcbs, i);</pre>
381
382
383
384
          char buffer[100];
385
          snprintf(buffer, \ size of (buffer), \ "%d\t%d\t%d\t%c\t%s\n", \ curr_pcb->pid,
386
                      curr_pcb->par_pid, curr_pcb->priority, curr_pcb->process_state,
387
                     curr_pcb->cmd_str);
         s_write(current_running_pcb->output_fd, buffer, strlen(buffer));
388
       }
389
390
       return NULL;
391 }
```

4.12.1.12 s_sleep()

```
void s_sleep ( \mbox{unsigned int } ticks \mbox{ )} \label{eq:condition}
```

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

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

Parameters

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

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

Definition at line 332 of file kern_sys_calls.c.

4.12.1.13 s_spawn()

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

Parameters

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

Returns

pid_t The process ID of the created child process or -1 on error

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

Definition at line 164 of file kern_sys_calls.c.

```
165
      pcb t* child:
      if (strcmp(argv[0], "shell") == 0) {
166
167
        child = k_proc_create(current_running_pcb, 0);
168
169
        child = k_proc_create(current_running_pcb, 1);
170
171
      if (child == NULL) {
  P_ERRNO = P_ENULL;
172
173
174
        return -1;
175
176
177
      spthread_t thread_handle;
178
      perror("Error in spthread_create in s_spawn call");
}
      if (spthread_create(&thread_handle, NULL, func, argv) != 0) {
179
180
181
182
183
      child->cmd_str = strdup(argv[0]);
      child->thread_handle = thread_handle;
184
      child->input_fd = fd0;
child->output_fd = fd1;
child->fd_table[0] = fd0;
185
186
187
188
      child->fd_table[1] = fd1;
189
      log_generic_event('C', child->pid, child->priority, child->cmd_str);
190
191
192
      return child->pid;
193 }
```

4.12.1.14 s_spawn_init()

```
pid_t s_spawn_init ( )
```

Similar to s_spawn except only called when you want to spawn the init process. It will create the init process and also spawn in the shell.

Returns

the pid_t of the created process on success or -1 on error

Similar to s_spawn except only called when you want to spawn the init process. It will create the init process and also spawn in the shell.

Definition at line 137 of file kern_sys_calls.c.

```
pcb_t* init = k_proc_create(NULL, 0);
if (init == NULL) {
   P_ERRNO = P_ENULL;
138
139
140
141
        return -1;
142
143
144
      spthread_t thread_handle;
      if (spthread_create(&thread_handle, NULL, init_func, NULL) != 0) {
145
       perror("Error in spthread_create in s_spawn_init call");
146
147
148
149
      init->cmd_str = strdup("init");
150
      init->thread_handle = thread_handle;
151
     return init->pid;
152 }
```

4.12.1.15 s_waitpid()

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

Parameters

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

Returns

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

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

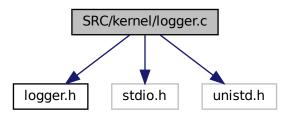
Definition at line 198 of file kern_sys_calls.c.

```
198
199
       pcb_t* parent = current_running_pcb;
200
       if (parent == NULL) {
2.01
        return -1;
202
203
204
       // if no children, return -1
205
       bool has_child = false;
206
       for (int i = 0; i < vec_len(&current_pcbs); i++) {</pre>
207
         pcb_t* child = vec_get(&current_pcbs, i);
         if (child->par_pid == parent->pid) {
  has_child = true;
208
209
210
           break;
211
212
213
       if (!has_child) {
         return -1;
214
215
216
       // Scan the zombie queue first for terminated children.
218
       for (int i = 0; i < vec_len(&zombie_queue); i++) {</pre>
219
         pcb_t* child = vec_get(&zombie_queue, i);
         if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
220
           if (wstatus != NULL) {
  *wstatus = child->process_status;
221
222
223
224
            \label{log_generic_event} \mbox{log\_generic\_event('W', child->pid, child->priority, child->cmd\_str);}
225
           vec_erase_no_deletor(&zombie_queue, i);
226
           delete_from_explicit_queue(&parent->child_pcbs, child->pid);
227
           k_proc_cleanup(child);
228
           return child->pid;
229
230
231
232
       // If nohang is true, return immediately if no child has exited
       if (nohang) {
233
234
        return 0;
235
236
237
       // Block the parent until a child exits
      delete_from_queue(parent->priority, parent->pid);
parent->process_state = 'B';
log_generic_event('B', parent->pid, parent->priority, parent->cmd_str);
238
239
240
241
242
       while (true) {
243
         // Scan the zombie queue first for terminated children.
244
         for (int i = 0; i < vec_len(&zombie_queue); i++) {</pre>
           pcb_t* child = vec_get(&zombie_queue, i);
if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
245
246
247
             if (wstatus != NULL) {
248
                *wstatus = child->process_status;
249
250
              \label{log_generic_event} $$\log_{\mathtt{generic_event}}('\,\mathtt{W'}$, child->pid, child->priority, child->cmd\_str)$;
             vec_erase_no_deletor(&zombie_queue, i);
delete_from_explicit_queue(&parent->child_pcbs, child->pid);
251
252
              k_proc_cleanup(child);
253
              return child->pid;
255
256
257
         // scan children of current running process for non-terminated state changes for (int i = 0; i < vec\_len(&parent->child\_pcbs); i++) {
258
259
          pcb_t* child = vec_get(&parent->child_pcbs, i);
260
            if ((pid == -1 || child->pid == pid) && (child->process_status == 21 || child->process_status ==
        23)) { // signaled
262
              if (wstatus != NULL) {
263
                *wstatus = child->process_status;
264
265
              log_generic_event('W', child->pid, child->priority, child->cmd_str);
              child->process_status = 0; // reset status
266
267
              return child->pid;
268
269
        }
270
      // If we get here, something went wrong
273
      return -1;
```

4.13 SRC/kernel/logger.c File Reference

#include "logger.h"

```
#include <stdio.h>
#include <unistd.h>
Include dependency graph for logger.c:
```



Functions

- void log_scheduling_event (int pid, int queue_num, char *process_name)

 Logs a scheduling event i.e. the scheduling of a process for this clock tick.
- void log_generic_event (char event_type, int pid, int nice_value, char *process_name)
 - Logs a non-nice, non-scheduling event (i.e any event that follows the EVENT PID NICE_VALUE PROCESS_NAME format)
- void log_nice_event (int pid, int old_nice_value, int new_nice_value, char *process_name)

 Logs a nice event, which is the adjusting of a process's nice value.

4.13.1 Function Documentation

4.13.1.1 log_generic_event()

Logs a non-nice, non-scheduling event (i.e any event that follows the EVENT PID NICE_VALUE PROCESS_NAME format)

Parameters

event_type	the type of event, defined by: 'C' = CREATE, 'S' = SIGNALED, 'E' = EXITED, 'Z' = ZOMBIE, 'O' = ORPHAN, 'W' = WAITED 'B' = BLOCKED, 'U' = UNBLOCKED 's' = STOPPED, 'c' = CONTINUED (notably lower-cased)
pid	process pid
nice_value	process nice value
process_name	string containing process name

Precondition

assumes event_type matches one of the above characters

Postcondition

will perror if the write fails

Definition at line 14 of file logger.c.

```
15
      char* operation;
16
17
      switch(event_type) {
          case 'C':
18
19
             operation = "CREATE";
         break; case 'S':
21
           operation = "SIGNALED";
22
         break; case 'E':
23
24
             operation = "EXITED";
25
26
             break;
27
         operation = "ZOMBIE";
break;
case '0':
28
29
30
         operation = "ORPHAN";
break;
case 'W':
31
34
            operation = "WAITED";
         break;
case 'B':
35
36
            operation = "BLOCKED";
37
38
             break;
          case 'U':
          operation = "UNBLOCKED";
40
         break; case 's':
41
42
             operation = "STOPPED";
43
44
             break;
45
          default:
46
             operation = "CONTINUED";
47
48
     }
49
      char buffer[200];
50
      pid, nice_value, process_name);
      if (write(log_fd, buffer, str_len) == -1) {
         perror("error in writing to the log file for generic event");
53
54
55 }
```

4.13.1.2 log_nice_event()

Logs a nice event, which is the adjusting of a process's nice value.

Parameters

pid	process pid
old_nice_value	old nice value
new_nice_value	new nice value
process_name	string containing process name

Generated by Doxygen

Postcondition

will perror if the write fails

Definition at line 57 of file logger.c.

```
char buffer[200];

char buffer[200];

int str_len = snprintf(buffer, sizeof(buffer), "[%d]\tNICE\t%d\t%d\t%d\t%s\n", tick_counter, pid, old_nice_value, new_nice_value, process_name);

if (write(log_fd, buffer, str_len) == -1) {
    perror("error in writing to the log file for nice event");
}

are char buffer[200];

[%d]\tNICE\t%d\t%d\t%d\t%s\n", tick_counter, pid, old_nice_value, new_nice_value, process_name);

if (write(log_fd, buffer, str_len) == -1) {
    perror("error in writing to the log file for nice event");
}
```

4.13.1.3 log_scheduling_event()

Logs a scheduling event i.e. the scheduling of a process for this clock tick.

Parameters

pid	pid of the process being scheduled
queue_num	the priority queue num of the process
process_name	string containing scheduled process's name

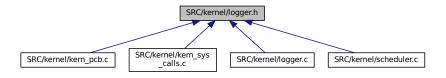
Postcondition

will perror if the write fails

Definition at line 6 of file logger.c.

4.14 SRC/kernel/logger.h File Reference

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



Functions

- void log_scheduling_event (int pid, int queue_num, char *process_name)
 - Logs a scheduling event i.e. the scheduling of a process for this clock tick.
- void log_generic_event (char event_type, int pid, int nice_value, char *process_name)
 - Logs a non-nice, non-scheduling event (i.e any event that follows the EVENT PID NICE_VALUE PROCESS_NAME format)
- void log_nice_event (int pid, int old_nice_value, int new_nice_value, char *process_name)
 - Logs a nice event, which is the adjusting of a process's nice value.

Variables

- · int tick_counter
- int log_fd

4.14.1 Function Documentation

4.14.1.1 log generic event()

Logs a non-nice, non-scheduling event (i.e any event that follows the EVENT PID NICE_VALUE PROCESS_NAME format)

Parameters

event_type	the type of event, defined by: 'C' = CREATE, 'S' = SIGNALED, 'E' = EXITED, 'Z' = ZOMBIE, 'O' = ORPHAN, 'W' = WAITED 'B' = BLOCKED, 'U' = UNBLOCKED 's' = STOPPED, 'c' = CONTINUED (notably lower-cased)
pid	process pid
nice_value	process nice value
process_name	string containing process name

Precondition

assumes event_type matches one of the above characters

Postcondition

will perror if the write fails

Definition at line 14 of file logger.c.

```
14
                                                                                                  {
15
       char* operation;
16
17
       switch(event_type) {
           case 'C':
18
                operation = "CREATE";
19
           break; case 'S':
20
21
              operation = "SIGNALED";
22
           break;
case 'E':
23
24
              operation = "EXITED";
25
26
           break; case 'Z':
           operation = "ZOMBIE";
28
           break; case 'O':
29
3.0
               operation = "ORPHAN";
31
32
                break;
           case 'W':
33
           operation = "WAITED";
       break;
case 'B':
35
36
        operation = "BLOCKED";
break;
case 'U':
37
38
          o :
   operation = "UNBLOCKED";
   break;
case 's':
40
41
42
            operation = "STOPPED";
43
44
                break;
45
           default:
46
              operation = "CONTINUED";
47
48
       }
49
       char buffer[200];
50
       int str_len = snprintf(buffer, sizeof(buffer), "[%d]\t%s\t%d\t%s\n", tick_counter, operation,
       pid, nice_value, process_name);
if (write(log_fd, buffer, str_len) == -1) {
52
53
           perror("error in writing to the log file for generic event");
54
55 }
```

4.14.1.2 log_nice_event()

Logs a nice event, which is the adjusting of a process's nice value.

Parameters

pid	process pid
old_nice_value	old nice value
new_nice_value	new nice value
process_name	string containing process name

Postcondition

will perror if the write fails

Definition at line 57 of file logger.c.

4.14.1.3 log_scheduling_event()

```
void log_scheduling_event (
          int pid,
          int queue_num,
          char * process_name )
```

Logs a scheduling event i.e. the scheduling of a process for this clock tick.

Parameters

pid	pid of the process being scheduled
queue_num	the priority queue num of the process
process_name	string containing scheduled process's name

Postcondition

will perror if the write fails

Definition at line 6 of file logger.c.

4.14.2 Variable Documentation

4.14.2.1 log_fd

```
int log_fd [extern]
```

Definition at line 36 of file scheduler.c.

4.14.2.2 tick_counter

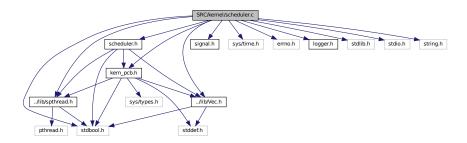
```
int tick_counter [extern]
```

Definition at line 35 of file scheduler.c.

4.15 SRC/kernel/scheduler.c File Reference

```
#include "scheduler.h"
#include <signal.h>
#include <stdbool.h>
#include <sys/time.h>
#include "../lib/Vec.h"
#include "../lib/spthread.h"
#include "errno.h"
#include "kern_pcb.h"
#include "logger.h"
#include "stdlib.h"
#include <stdio.h>
#include <string.h>
```

Include dependency graph for scheduler.c:



Functions

• void initialize_scheduler_queues ()

Initializes the scheduler queues.

• void free_scheduler_queues ()

Frees the scheduler queues.

• int generate_next_priority ()

Generates the next priority for scheduling based on the defined probabilities.

pcb_t * get_next_pcb (int priority)

Gets the next PCB from the specified priority queue.

void put pcb into correct queue (pcb t*pcb)

Puts the given PCB into the correct queue based on its priority and state.

void delete_process_from_particular_queue (pcb_t *pcb, Vec *queue)

Deletes the given PCB from the specified queue.

void delete process from all queues except current (pcb t *pcb)

Deletes the given PCB from all queues except the current one.

void delete_process_from_all_queues (pcb_t *pcb)

Deletes the given PCB from all queues.

• pcb_t * get_pcb_in_queue (Vec *queue, pid_t pid)

Gets the PCB with the specified PID from the given queue.

bool child_in_zombie_queue (pcb_t *parent)

Checks if the given parent PCB has any children in the zombie queue.

bool child_with_changed_process_status (pcb_t *parent)

Checks if the given parent PCB has any children with a changed process status.

void alarm handler (int signum)

Signal handler for SIGALRM.

• void handle_signal (pcb_t *pcb, int signal)

Handles the specified signal for the given PCB.

void s_shutdown_pennos (void)

Shuts down the scheduler and cleans up resources.

• void scheduler ()

The main scheduler function for PennOS.

Variables

- Vec zero_priority_queue
- · Vec one_priority_queue
- Vec two_priority_queue
- · Vec zombie queue
- Vec sleep_blocked_queue
- Vec current_pcbs
- int tick_counter = 0
- int log_fd
- pcb_t * current_running_pcb
- int curr_priority_arr_index = 0
- int det_priorities_arr [19]

4.15.1 Function Documentation

4.15.1.1 alarm_handler()

Signal handler for SIGALRM.

Handles the alarm signal.

Definition at line 225 of file scheduler.c.

```
225
226 tick_counter++;
```

4.15.1.2 child_in_zombie_queue()

Checks if the given parent PCB has any children in the zombie queue.

Checks if a child of the given parent process is in the zombie queue.

Definition at line 199 of file scheduler.c.

```
199
200    for (int i = 0; i < vec_len(&zombie_queue); i++) {
201        pcb_t* child = vec_get(&zombie_queue, i);
202        if (child->par_pid == parent->pid) {
203            return true;
204        }
205     }
206     return false;
207 }
```

4.15.1.3 child_with_changed_process_status()

Checks if the given parent PCB has any children with a changed process status.

Checks if a child of the given parent process has a changed process status.

Definition at line 212 of file scheduler.c.

4.15.1.4 delete_process_from_all_queues()

```
void delete_process_from_all_queues (
          pcb_t * pcb )
```

Deletes the given PCB from all queues.

Searches through all of the scheduler's queues and deletes the the given pcb from all of them. Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.

Definition at line 177 of file scheduler.c.

```
177
delete_process_from_all_queues_except_current(pcb);
delete_process_from_particular_queue(pcb, &current_pcbs);
180 }
```

4.15.1.5 delete_process_from_all_queues_except_current()

Deletes the given PCB from all queues except the current one.

Searches through all of the scheduler's queues except the one containing all of the current processes and deletes the given pcb from all of them Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.

Definition at line 166 of file scheduler.c.

```
166
167 delete_process_from_particular_queue(pcb, &zero_priority_queue);
168 delete_process_from_particular_queue(pcb, &one_priority_queue);
169 delete_process_from_particular_queue(pcb, &two_priority_queue);
170 delete_process_from_particular_queue(pcb, &zombie_queue);
171 delete_process_from_particular_queue(pcb, &sleep_blocked_queue);
172 }
```

4.15.1.6 delete_process_from_particular_queue()

Deletes the given PCB from the specified queue.

Given a queue in the form of a vector, searches through it for the given pcb and deletes it from the queue if found. Notably, it does not free the pcb. Instead, the implmentation calls vec_erase_no_deletor instead of vec_erase. If the pcb isn't in the queue, this function does nothing.

Definition at line 153 of file scheduler.c.

```
153
154
      for (int i = 0; i < vec_len(queue); i++) {</pre>
       pcb_t* curr_pcb = vec_get(queue, i);
155
156
        if (curr_pcb->pid == pcb->pid)
157
         vec_erase_no_deletor(queue, i);
158
         return;
159
       }
160
    }
161 }
```

4.15.1.7 free_scheduler_queues()

```
void free_scheduler_queues ( )
```

Frees the scheduler queues.

Frees the scheduler queues. This function should be called when the scheduler is no longer needed.

Definition at line 66 of file scheduler.c.

4.15.1.8 generate_next_priority()

```
int generate_next_priority ( )
```

Generates the next priority for scheduling based on the defined probabilities.

Deterministically chooses an integer from 0, 1, 2 at the prescribed probabilites. In particular, 0 is output 1.5x more than 1, which is output 1.5x more than 2. Notably, it accounts for cases where some of the queues are empty. If all queues are empty, it'll return -1.

Definition at line 82 of file scheduler.c.

```
83
      // check if all queues are empty
84
     if (vec_is_empty(&zero_priority_queue) && vec_is_empty(&one_priority_queue) &&
85
          vec_is_empty(&two_priority_queue)) {
86
        return -1;
     int priorities_attempted = 0;
     while (priorities_attempted < 19) {
  int curr_pri = det_priorities_arr[curr_priority_arr_index];</pre>
90
91
       curr_priority_arr_index = (curr_priority_arr_index + 1) % 19;
if (curr_pri == 0 && !vec_is_empty(&zero_priority_queue)) {
92
93
          priorities_attempted++;
       return 0;
} else if (curr_pri == 1 && !vec_is_empty(&one_priority_queue)) {
96
97
          priorities_attempted++;
98
          return 1:
       } else if (curr_pri == 2 && !vec_is_empty(&two_priority_queue)) {
99
         priorities_attempted++;
100
101
           return 2;
102
103
104
105
      return -1; // should never reach
```

4.15.1.9 get_next_pcb()

Gets the next PCB from the specified priority queue.

Returns the next PCB in the queue of the specified priority. or NULL if that queue is empty. Notably, it removes the PCB from the queue.

Definition at line 111 of file scheduler.c.

```
111
       if (priority == -1) { // all queues empty
112
113
        return NULL;
114
115
116
      pcb_t* next_pcb = NULL;
      if (priority == 0) {
  next_pcb = vec_get(&zero_priority_queue, 0);
117
118
        vec_erase_no_deletor(&zero_priority_queue, 0);
119
      } else if (priority == 1) {
120
121
       next_pcb = vec_get(&one_priority_queue, 0);
122
        vec_erase_no_deletor(&one_priority_queue, 0);
      } else if (priority == 2) {
  next_pcb = vec_get(&two_priority_queue, 0);
123
124
        vec_erase_no_deletor(&two_priority_queue, 0);
125
126
127
128
      return next_pcb;
129 }
```

4.15.1.10 get_pcb_in_queue()

Gets the PCB with the specified PID from the given queue.

Given a queue, searches for a particular pid inside that queue and, if found, returns the pcb_t* associated with that pid.

Definition at line 185 of file scheduler.c.

4.15.1.11 handle_signal()

Handles the specified signal for the given PCB.

Handles a signal for a given process.

Definition at line 232 of file scheduler.c.

```
232
233
        switch (signal)
234
          case 0: // P_SIGSTOP
            if (pcb->process_state == 'R' || pcb->process_state == 'B') {
  pcb->process_state = 'S';
235
236
               pcb->process_state = 0;
log_generic_event('s', pcb->pid, pcb->priority, pcb->cmd_str);
delete_process_from_all_queues_except_current(pcb);
237
238
               pcb->process_status = 21; // STOPPED_BY_SIG
240
241
            pcb->signals[0] = false;
242
            break;
                                                         // P_SIGCONT
243
          case 1:
            if (pcb->process_state == 'S') { // Only continue if stopped
244
               pcb->process_state = 'R';
245
               log_generic_event('c', pcb->pid, pcb->priority, pcb->cmd_str);
delete_process_from_all_queues_except_current(pcb);
246
247
248
               put_pcb_into_correct_queue(pcb);
249
               pcb->process_status = 23; // Reset status
250
251
            pcb->signals[1] = false;
252
             break;
253
                                                          // P_SIGTERM
           if (pcb->process_state != 'Z') { // Don't terminate if already zombie
  pcb->process_state = 'Z';
  pcb->process_status = 22; // TERM_BY_SIG
254
255
256
               log_generic_event('Z', pcb->pid, pcb->priority, pcb->cmd_str);
delete_process_from_all_queues_except_current(pcb);
257
258
259
               put_pcb_into_correct_queue(pcb);
260
               pcb->process_status = 22; // TERM_BY_SIG
2.61
             pcb->signals[2] = false;
262
263
             break:
264
       }
265 }
```

4.15.1.12 initialize_scheduler_queues()

```
void initialize_scheduler_queues ( )
```

Initializes the scheduler queues.

Initializes the scheduler queues. This function should be called before any other scheduler functions are called.

Note

The deconstructors for the queues are set to NULL to prevent double freeing when exiting PennOS.

Definition at line 54 of file scheduler.c.

4.15.1.13 put_pcb_into_correct_queue()

Puts the given PCB into the correct queue based on its priority and state.

Puts the given pcb struct pointer into its appropriate queue. Notably, it solely uses the pcb's interal fields to determine the correct queue (priority and state).

Definition at line 134 of file scheduler.c.

```
135
      if (pcb->process_state == 'R') {
136
       if (pcb->priority == 0) {
       vec_push_back(&zero_priority_queue, pcb);
} else if (pcb->priority == 1) {
137
138
139
         vec_push_back(&one_priority_queue, pcb);
       } else if (pcb->priority == 2) {
141
          vec_push_back(&two_priority_queue, pcb);
142
143
      } else if (pcb->process_state == 'Z') {
      vec_push_back(&zombie_queue, pcb);
} else if (pcb->process_state == 'B' || pcb->is_sleeping) {
144
145
146
        vec_push_back(&sleep_blocked_queue, pcb);
147
148 }
```

4.15.1.14 s shutdown pennos()

Shuts down the scheduler and cleans up resources.

Shuts down the PennOS scheduler.

Definition at line 270 of file scheduler.c.

```
270 {
271     scheduling_done = true;
272 }
```

4.15.1.15 scheduler()

```
void scheduler ( )
```

The main scheduler function for PennOS.

This function manages process scheduling, signal handling, and timer-based preemption. It ensures that processes are executed based on their priority and handles signals for both the currently running process and other processes.

Definition at line 277 of file scheduler.c.

```
int curr_priority_queue_num;
280
      // mask for while scheduler is waiting for alarm
281
      sigset_t suspend_set;
282
      sigfillset(&suspend_set);
      sigdelset(&suspend_set, SIGALRM);
283
284
285
      // ensure sigarlm doesn't terminate the process
      struct sigaction act = (struct sigaction) {
287
          .sa_handler = alarm_handler,
288
           .sa_mask = suspend_set,
289
           .sa_flags = SA_RESTART,
290
291
      sigaction(SIGALRM, &act, NULL);
292
293
      // make sure SIGALRM is unblocked
294
      sigset_t alarm_set;
295
      sigemptyset(&alarm_set);
296
      sigaddset (&alarm set, SIGALRM);
      pthread_sigmask(SIG_UNBLOCK, &alarm_set, NULL);
297
298
299
      struct itimerval it;
300
      it.it_interval = (struct timeval) {.tv_usec = hundred_millisec};
301
      it.it_value = it.it_interval;
      setitimer(ITIMER_REAL, &it, NULL);
302
303
304
      while (!scheduling_done) {
305
        // handle signals for the currently running process
306
        if (current_running_pcb != NULL) {
307
           for (int i = 0; i < 3; i++) {
             if (current_running_pcb->signals[i]) {
308
              handle_signal(current_running_pcb, i);
309
               // If process was terminated, don't continue scheduling it
310
311
               if (current_running_pcb->process_state != 'R') {
312
                 current_running_pcb = NULL;
313
                 break;
314
315
            }
          }
316
318
319
        // handle signals for all other processes (currently running or not)
        for (int i = 0; i < vec_len(&current_pcbs); i++) {
  pcb_t* curr_pcb = vec_get(&current_pcbs, i);</pre>
320
321
          for (int j = 0; j < 3; j++) {
   if (curr_pcb->signals[j]) {
322
323
324
              handle_signal(curr_pcb, j);
325
326
          }
327
328
329
        // Check sleep/blocked queue to move processes back to scheduable queues
330
        for (int i = 0; i < vec_len(&sleep_blocked_queue); i++) {</pre>
331
          pcb_t* blocked_proc = vec_get(&sleep_blocked_queue, i);
          bool make_runnable = false;
if (blocked_proc->is_sleeping &&
332
333
               blocked_proc->time_to_wake == tick_counter) {
334
335
             blocked_proc->is_sleeping = false;
336
             blocked_proc->time_to_wake = -1;
337
             blocked_proc->signals[2] = false; // Unlikely, but reset signal
338
            make_runnable = true;
339
          } else if (blocked_proc->is_sleeping &&
                    blocked_proc->signals[2]) { // P_SIGTERM received
340
341
             blocked_proc->is_sleeping = false;
342
             blocked_proc->process_state = 'Z';
343
             blocked_proc->process_status = 22; // TERM_BY_SIG
344
             blocked_proc->signals[2] = false;
345
            delete_process_from_all_queues_except_current(blocked_proc);
             put_pcb_into_correct_queue(blocked_proc);
log_generic_event('Z', blocked_proc->pid, blocked_proc->priority,
346
347
                                blocked_proc->cmd_str);
```

```
i--;
350
          } else if (child_in_zombie_queue(blocked_proc)) {
351
            make_runnable = true;
          } else if (child_with_changed_process_status(blocked_proc)) {
352
353
           make_runnable = true;
354
355
356
          if (make_runnable) {
357
          blocked_proc->process_state = 'R';
358
            vec_erase_no_deletor(&sleep_blocked_queue, i);
            delete_process_from_all_queues_except_current(blocked_proc);
359
            put_pcb_into_correct_queue(blocked_proc);
log_generic_event('U', blocked_proc->pid, blocked_proc->priority,
360
361
362
                               blocked_proc->cmd_str);
363
364
365
366
367
        curr_priority_queue_num = generate_next_priority();
368
369
        current_running_pcb = get_next_pcb(curr_priority_queue_num);
370
        if (current_running_pcb == NULL) {
371
        sigsuspend(&suspend_set); // idle until signal received
372
          continue;
373
374
375
        log_scheduling_event(current_running_pcb->pid, curr_priority_queue_num,
376
                              current_running_pcb->cmd_str);
377
378
        if (spthread_continue(current_running_pcb->thread_handle) != 0 &&
379
            errno != EINTR) {
380
         perror("spthread_continue failed in scheduler");
381
382
        sigsuspend(&suspend_set);
        if (spthread_suspend(current_running_pcb->thread_handle) != 0 &&
    errno != EINTR) {
383
384
385
          perror("spthread_suspend failed in scheduler");
386
387
        put_pcb_into_correct_queue(current_running_pcb);
388
389 }
```

4.15.2 Variable Documentation

4.15.2.1 curr priority arr index

int curr_priority_arr_index = 0

Definition at line 40 of file scheduler.c.

4.15.2.2 current pcbs

Vec current_pcbs

Definition at line 30 of file scheduler.c.

4.15.2.3 current_running_pcb

```
pcb_t* current_running_pcb
```

Definition at line 38 of file scheduler.c.

4.15.2.4 det_priorities_arr

```
int det_priorities_arr[19]
```

Initial value:

```
Initial value:
= {0, 1, 2, 0, 0, 1, 0, 1, 2, 0,
0, 1, 2, 0, 1, 0, 0, 1, 2}
```

Definition at line 41 of file scheduler.c.

4.15.2.5 log_fd

```
int log_fd
```

Definition at line 36 of file scheduler.c.

4.15.2.6 one_priority_queue

```
Vec one_priority_queue
```

Definition at line 25 of file scheduler.c.

4.15.2.7 sleep_blocked_queue

```
Vec sleep_blocked_queue
```

Definition at line 28 of file scheduler.c.

4.15.2.8 tick_counter

```
int tick\_counter = 0
```

Definition at line 35 of file scheduler.c.

4.15.2.9 two_priority_queue

```
Vec two_priority_queue
```

Definition at line 26 of file scheduler.c.

4.15.2.10 zero priority queue

```
Vec zero_priority_queue
```

Definition at line 24 of file scheduler.c.

4.15.2.11 zombie_queue

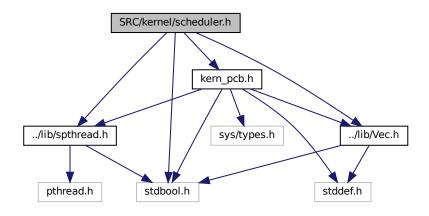
```
Vec zombie_queue
```

Definition at line 27 of file scheduler.c.

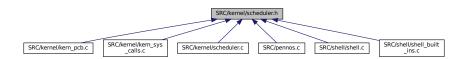
4.16 SRC/kernel/scheduler.h File Reference

```
#include <stdbool.h>
#include "../lib/Vec.h"
#include "../lib/spthread.h"
#include "kern_pcb.h"
```

Include dependency graph for scheduler.h:



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



Functions

· void initialize scheduler queues ()

Initializes the scheduler queues. This function should be called before any other scheduler functions are called.

void free scheduler queues ()

Frees the scheduler queues. This function should be called when the scheduler is no longer needed.

• int generate next priority ()

Deterministically chooses an integer from 0, 1, 2 at the prescribed probabilites. In particular, 0 is output 1.5x more than 1, which is output 1.5x more than 2. Notably, it accounts for cases where some of the queues are empty. If all queues are empty, it'll return -1.

pcb t * get next pcb (int priority)

Returns the next PCB in the queue of the specified priority. or NULL if that queue is empty. Notably, it removes the PCB from the queue.

void put pcb into correct queue (pcb t*pcb)

Puts the given pcb struct pointer into its appropriate queue. Notably, it solely uses the pcb's interal fields to determine the correct queue (priority and state).

void delete_process_from_particular_queue (pcb_t *pcb, Vec *queue)

Given a queue in the form of a vector, searches through it for the given pcb and deletes it from the queue if found. Notably, it does not free the pcb. Instead, the implmentation calls vec_erase_no_deletor instead of vec_erase. If the pcb isn't in the queue, this function does nothing.

void delete_process_from_all_queues_except_current (pcb_t *pcb)

Searches through all of the scheduler's queues except the one containing all of the current processes and deletes the given pcb from all of them Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.

void delete_process_from_all_queues (pcb_t *pcb)

Searches through all of the scheduler's queues and deletes the the given pcb from all of them. Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.

pcb t * get pcb in queue (Vec *queue, pid t pid)

Given a queue, searches for a particular pid inside that queue and, if found, returns the pcb_t* associated with that pid.

bool child in zombie queue (pcb t *parent)

Checks if a child of the given parent process is in the zombie queue.

bool child_with_changed_process_status (pcb_t *parent)

Checks if a child of the given parent process has a changed process status.

• void alarm_handler (int signum)

Handles the alarm signal.

void handle_signal (pcb_t *pcb, int signal)

Handles a signal for a given process.

• void scheduler ()

The main scheduler function for PennOS.

void s_shutdown_pennos ()

Shuts down the PennOS scheduler.

4.16.1 Function Documentation

4.16.1.1 alarm_handler()

```
void alarm_handler ( int \ \textit{signum} \ )
```

Handles the alarm signal.

This function is triggered when the alarm signal is received. It increments the global tick counter, which is used for scheduling and timing purposes.

Parameters

used in this implementation).	signum The signal number
-------------------------------	--------------------------

Handles the alarm signal.

Definition at line 225 of file scheduler.c.

4.16.1.2 child_in_zombie_queue()

Checks if a child of the given parent process is in the zombie queue.

This function iterates through the zombie queue to determine if any process in the queue has the given parent process as its parent.

Parameters

parent	A pointer to the parent PCB.
--------	------------------------------

Returns

true if a child of the parent is in the zombie queue, false otherwise.

Checks if a child of the given parent process is in the zombie queue.

Definition at line 199 of file scheduler.c.

```
199
200     for (int i = 0; i < vec_len(&zombie_queue); i++) {
201         pcb_t* child = vec_get(&zombie_queue, i);
202         if (child->par_pid == parent->pid) {
203             return true;
204         }
205         }
206         return false;
207 }
```

4.16.1.3 child_with_changed_process_status()

Checks if a child of the given parent process has a changed process status.

This function iterates through the current PCBs to determine if any child of the given parent process has a non-zero process status, indicating a change.

Parameters

parent	A pointer to the parent PCB.
--------	------------------------------

Returns

true if a child of the parent has a changed process status, false otherwise.

Checks if a child of the given parent process has a changed process status.

Definition at line 212 of file scheduler.c.

4.16.1.4 delete process from all queues()

```
void delete_process_from_all_queues (
          pcb_t * pcb )
```

Searches through all of the scheduler's queues and deletes the the given pcb from all of them. Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.

Parameters

```
pcb a pointer to the pcb with the pid to delete
```

Searches through all of the scheduler's queues and deletes the the given pcb from all of them. Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.

Definition at line 177 of file scheduler.c.

```
177 {
178 delete_process_from_all_queues_except_current(pcb);
179 delete_process_from_particular_queue(pcb, &current_pcbs);
180 }
```

4.16.1.5 delete_process_from_all_queues_except_current()

```
void delete_process_from_all_queues_except_current ( pcb\_t \ * \ pcb \ )
```

Searches through all of the scheduler's queues except the one containing all of the current processes and deletes the given pcb from all of them Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.

Parameters

pcb a pointer to the pcb with the pid to delete

Searches through all of the scheduler's queues except the one containing all of the current processes and deletes the given pcb from all of them Notably, it does not free the pcb via calling vec_erase_no_deletor instead of vec_erase. If a particular queue does not contain the pcb, nothing occurs.

Definition at line 166 of file scheduler.c.

```
delete_process_from_particular_queue(pcb, &zero_priority_queue);
delete_process_from_particular_queue(pcb, &one_priority_queue);
delete_process_from_particular_queue(pcb, &two_priority_queue);
delete_process_from_particular_queue(pcb, &zombie_queue);
delete_process_from_particular_queue(pcb, &sleep_blocked_queue);
171
delete_process_from_particular_queue(pcb, &sleep_blocked_queue);
```

4.16.1.6 delete_process_from_particular_queue()

Given a queue in the form of a vector, searches through it for the given pcb and deletes it from the queue if found. Notably, it does not free the pcb. Instead, the implmentation calls vec_erase_no_deletor instead of vec_erase. If the pcb isn't in the queue, this function does nothing.

Given a queue in the form of a vector, searches through it for the given pcb and deletes it from the queue if found. Notably, it does not free the pcb. Instead, the implmentation calls vec_erase_no_deletor instead of vec_erase. If the pcb isn't in the queue, this function does nothing.

Definition at line 153 of file scheduler.c.

4.16.1.7 free_scheduler_queues()

```
void free_scheduler_queues ( )
```

Frees the scheduler queues. This function should be called when the scheduler is no longer needed.

Frees the scheduler queues. This function should be called when the scheduler is no longer needed.

Definition at line 66 of file scheduler.c.

4.16.1.8 generate_next_priority()

```
int generate_next_priority ( )
```

Deterministically chooses an integer from 0, 1, 2 at the prescribed probabilites. In particular, 0 is output 1.5x more than 1, which is output 1.5x more than 2. Notably, it accounts for cases where some of the queues are empty. If all queues are empty, it'll return -1.

Precondition

assumes that at least one of the scheduler queues in non-empty

Returns

int 0, 1, or 2 for priority or -1 to signify that all queues are empty

Deterministically chooses an integer from 0, 1, 2 at the prescribed probabilites. In particular, 0 is output 1.5x more than 1, which is output 1.5x more than 2. Notably, it accounts for cases where some of the queues are empty. If all queues are empty, it'll return -1.

Definition at line 82 of file scheduler.c.

```
// check if all queues are empty
83
84
                       \begin{tabular}{ll} if (vec\_is\_empty(\&zero\_priority\_queue) &\& vec\_is\_empty(\&one\_priority\_queue) &\& vec\_is\_empty(\&one\_pr
85
                                        vec_is_empty(&two_priority_queue)) {
                               return -1;
87
88
89
                     int priorities_attempted = 0;
                    while (priorities_attempted < 19) {</pre>
90
                             int curr_pri = det_priorities_arr[curr_priority_arr_index];
                             curr_priority_arr_index = (curr_priority_arr_index + 1) % 19;
if (curr_pri == 0 && !vec_is_empty(&zero_priority_queue)) {
94
                                priorities_attempted++;
95
                             } else if (curr_pri == 1 && !vec_is_empty(&one_priority_queue)) {
96
97
                                     priorities_attempted++;
98
                                        return 1;
                             } else if (curr_pri == 2 && !vec_is_empty(&two_priority_queue)) {
100
                                           priorities_attempted++;
101
                                              return 2;
102
103
                        }
104
                         return -1; // should never reach
```

4.16.1.9 get_next_pcb()

Returns the next PCB in the queue of the specified priority. or NULL if that queue is empty. Notably, it removes the PCB from the queue.

Parameters

priority queue priority to get next PCB from, or -1 if none

Returns

a ptr to the next pcb struct in queue or NULL if the queue is empty

Returns the next PCB in the queue of the specified priority. or NULL if that queue is empty. Notably, it removes the PCB from the queue.

Definition at line 111 of file scheduler.c.

```
111
       if (priority == -1) { // all queues empty
112
         return NULL;
113
114
115
116
       pcb_t* next_pcb = NULL;
       if (priority == 0) {
  next_pcb = vec_get(&zero_priority_queue, 0);
  vec_erase_no_deletor(&zero_priority_queue, 0);
117
118
119
120
       } else if (priority == 1) {
        next_pcb = vec_get(&one_priority_queue, 0);
         vec_erase_no_deletor(&one_priority_queue, 0);
122
123
       } else if (priority == 2) {
       next_pcb = vec_get(&two_priority_queue, 0);
vec_erase_no_deletor(&two_priority_queue, 0);
124
125
126
127
128
       return next_pcb;
129 }
```

4.16.1.10 get_pcb_in_queue()

Given a queue, searches for a particular pid inside that queue and, if found, returns the pcb_t* associated with that pid.

Parameters

queue	the queue of pcb_t* ptrs to search
pid	the pid to search for

Returns

a ptr to the pcb w/ the desired pid if found, NULL otherwise

Given a queue, searches for a particular pid inside that queue and, if found, returns the pcb_t* associated with that pid.

Definition at line 185 of file scheduler.c.

4.16.1.11 handle_signal()

Handles a signal for a given process.

This function processes a signal sent to a process and updates its state accordingly. Supported signals include:

- P_SIGSTOP: Stops the process.
- · P SIGCONT: Continues a stopped process.
- P_SIGTERM: Terminates the process.

Parameters

pcb	A pointer to the PCB of the process receiving the signal.
signal	The signal to handle (0 for P_SIGSTOP, 1 for P_SIGCONT, 2 for P_SIGTERM).

Handles a signal for a given process.

Definition at line 232 of file scheduler.c.

```
switch (signal)
233
234
            case 0: // P_SIGSTOP
             if (pcb->process_state == 'R' || pcb->process_state == 'B') {
   pcb->process_state = 'S';
235
236
                  log_generic_event('s', pcb->pid, pcb->priority, pcb->cmd_str);
delete_process_from_all_queues_except_current(pcb);
237
238
239
                  pcb->process_status = 21; // STOPPED_BY_SIG
240
241
               pcb->signals[0] = false;
242
               break;
243
                                                                     // P_SIGCONT
            case 1:
            if (pcb->process_state == 'S') { // Only continue if stopped
244
245
                 pcb->process_state = 'R';
                  log_generic_event('c', pcb->pid, pcb->priority, pcb->cmd_str);
delete_process_from_all_queues_except_current(pcb);
246
247
                  put_pcb_into_correct_queue(pcb);
pcb->process_status = 23; // Reset status
2.48
249
250
251
               pcb->signals[1] = false;
252
253
                                                                     // P SIGTERM
            if (pcb->process_state != 'Z') {    // Pon't terminate if already zombie
    pcb->process_state = 'Z';
    pcb->process_status = 22;    // TERM_BY_SIG
    log_generic_event('Z', pcb->pid, pcb->priority, pcb->cmd_str);
    delete_process_from_all_queues_except_current(pcb);

process_from_all_queues_except_current(pcb);
254
255
256
257
258
                 put_pcb_into_correct_queue(pcb);
pcb->process_status = 22; // TERM_BY_SIG
259
260
261
262
               pcb->signals[2] = false;
               break;
264
        }
265 }
```

4.16.1.12 initialize_scheduler_queues()

```
void initialize_scheduler_queues ( )
```

Initializes the scheduler queues. This function should be called before any other scheduler functions are called. Initializes the scheduler queues. This function should be called before any other scheduler functions are called.

Note

The deconstructors for the queues are set to NULL to prevent double freeing when exiting PennOS.

Definition at line 54 of file scheduler.c.

4.16.1.13 put_pcb_into_correct_queue()

Puts the given pcb struct pointer into its appropriate queue. Notably, it solely uses the pcb's interal fields to determine the correct queue (priority and state).

Puts the given pcb struct pointer into its appropriate queue. Notably, it solely uses the pcb's interal fields to determine the correct queue (priority and state).

Definition at line 134 of file scheduler.c.

```
135
      if (pcb->process_state == 'R') {
        if (pcb->priority == 0) {
136
        vec_push_back(&zero_priority_queue, pcb);
} else if (pcb->priority == 1) {
137
138
          vec_push_back(&one_priority_queue, pcb);
139
       } else if (pcb->priority == 2) {
140
141
           vec_push_back(&two_priority_queue, pcb);
142
      } else if (pcb->process_state == 'Z') {
143
      vec_push_back(&zombie_queue, pcb);
} else if (pcb->process_state == 'B' || pcb->is_sleeping) {
144
145
        vec_push_back(&sleep_blocked_queue, pcb);
147
148 }
```

4.16.1.14 s_shutdown_pennos()

Shuts down the PennOS scheduler.

This function sets the scheduling_done flag to true, signaling the scheduler to terminate its loop and shut down.

Shuts down the PennOS scheduler.

Definition at line 270 of file scheduler.c.

```
270
271    scheduling_done = true;
272 }
```

4.16.1.15 scheduler()

```
void scheduler ( )
```

The main scheduler function for PennOS.

This function manages process scheduling, signal handling, and timer-based preemption. It ensures that processes are executed based on their priority and handles signals for both the currently running process and other processes.

Definition at line 277 of file scheduler.c.

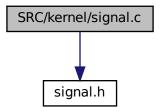
```
int curr_priority_queue_num;
280
      // mask for while scheduler is waiting for alarm
281
      sigset_t suspend_set;
282
      sigfillset(&suspend_set);
      sigdelset(&suspend_set, SIGALRM);
283
284
285
      // ensure sigarlm doesn't terminate the process
      struct sigaction act = (struct sigaction) {
287
          .sa_handler = alarm_handler,
288
           .sa_mask = suspend_set,
289
           .sa_flags = SA_RESTART,
290
291
      sigaction(SIGALRM, &act, NULL);
292
293
      // make sure SIGALRM is unblocked
294
      sigset_t alarm_set;
295
      sigemptyset(&alarm_set);
296
      sigaddset (&alarm set, SIGALRM);
      pthread_sigmask(SIG_UNBLOCK, &alarm_set, NULL);
297
298
299
      struct itimerval it;
300
      it.it_interval = (struct timeval) {.tv_usec = hundred_millisec};
301
      it.it_value = it.it_interval;
      setitimer(ITIMER_REAL, &it, NULL);
302
303
304
      while (!scheduling_done) {
305
        // handle signals for the currently running process
306
        if (current_running_pcb != NULL) {
307
           for (int i = 0; i < 3; i++) {
             if (current_running_pcb->signals[i]) {
308
              handle_signal(current_running_pcb, i);
309
               // If process was terminated, don't continue scheduling it
310
311
               if (current_running_pcb->process_state != 'R') {
312
                 current_running_pcb = NULL;
313
                 break;
314
315
            }
          }
316
318
319
        // handle signals for all other processes (currently running or not)
        for (int i = 0; i < vec_len(&current_pcbs); i++) {
  pcb_t* curr_pcb = vec_get(&current_pcbs, i);</pre>
320
321
          for (int j = 0; j < 3; j++) {
   if (curr_pcb->signals[j]) {
322
323
324
              handle_signal(curr_pcb, j);
325
326
          }
327
328
329
        // Check sleep/blocked queue to move processes back to scheduable queues
330
        for (int i = 0; i < vec_len(&sleep_blocked_queue); i++) {</pre>
331
          pcb_t* blocked_proc = vec_get(&sleep_blocked_queue, i);
          bool make_runnable = false;
if (blocked_proc->is_sleeping &&
332
333
               blocked_proc->time_to_wake == tick_counter) {
334
335
             blocked_proc->is_sleeping = false;
336
             blocked_proc->time_to_wake = -1;
337
             blocked_proc->signals[2] = false; // Unlikely, but reset signal
338
            make_runnable = true;
339
          } else if (blocked_proc->is_sleeping &&
                    blocked_proc->signals[2]) { // P_SIGTERM received
340
341
             blocked_proc->is_sleeping = false;
342
             blocked_proc->process_state = 'Z';
343
             blocked_proc->process_status = 22; // TERM_BY_SIG
344
             blocked_proc->signals[2] = false;
345
            delete_process_from_all_queues_except_current(blocked_proc);
             put_pcb_into_correct_queue(blocked_proc);
log_generic_event('Z', blocked_proc->pid, blocked_proc->priority,
346
347
                                blocked_proc->cmd_str);
```

```
i--;
350
          } else if (child_in_zombie_queue(blocked_proc)) {
351
            make_runnable = true;
          } else if (child_with_changed_process_status(blocked_proc)) {
352
353
            make_runnable = true;
354
355
356
          if (make_runnable) {
357
           blocked_proc->process_state = 'R';
358
            vec_erase_no_deletor(&sleep_blocked_queue, i);
            delete_process_from_all_queues_except_current(blocked_proc);
359
            put_pcb_into_correct_queue(blocked_proc);
log_generic_event('U', blocked_proc->pid, blocked_proc->priority,
360
361
362
                               blocked_proc->cmd_str);
363
364
365
366
367
        curr_priority_queue_num = generate_next_priority();
368
369
        current_running_pcb = get_next_pcb(curr_priority_queue_num);
370
        if (current_running_pcb == NULL) {
371
          sigsuspend(&suspend_set); // idle until signal received
372
          continue;
373
374
375
        log_scheduling_event(current_running_pcb->pid, curr_priority_queue_num,
376
                              current_running_pcb->cmd_str);
377
378
        if (spthread_continue(current_running_pcb->thread_handle) != 0 &&
379
            errno != EINTR) {
380
          perror("spthread_continue failed in scheduler");
381
382
        sigsuspend(&suspend_set);
        if (spthread_suspend(current_running_pcb->thread_handle) != 0 &&
    errno != EINTR) {
383
384
385
          perror("spthread_suspend failed in scheduler");
386
        put_pcb_into_correct_queue(current_running_pcb);
388
389 }
```

4.17 SRC/kernel/signal.c File Reference

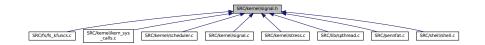
#include <signal.h>

Include dependency graph for signal.c:



4.18 SRC/kernel/signal.h File Reference

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



Macros

• #define P_SIGSTOP 0

Signals for PennOS.

- #define P_SIGCONT 1
- #define P_SIGTERM 2
- #define EXITED NORMALLY 20

Status definitions.

- #define STOPPED_BY_SIG 21
- #define TERM_BY_SIG 22
- #define CONT BY SIG 23
- #define P_WIFEXITED(status) ((status) == EXITED_NORMALLY)

User-level macros for waitpid status.

- #define P_WIFSTOPPED(status) ((status) == STOPPED_BY_SIG)
- #define P_WIFSIGNALED(status) ((status) == TERM_BY_SIG)

4.18.1 Macro Definition Documentation

4.18.1.1 CONT_BY_SIG

#define CONT_BY_SIG 23

Definition at line 17 of file signal.h.

4.18.1.2 EXITED_NORMALLY

#define EXITED_NORMALLY 20

Status definitions.

Definition at line 14 of file signal.h.

4.18.1.3 P_SIGCONT

```
#define P_SIGCONT 1
```

Definition at line 8 of file signal.h.

4.18.1.4 P_SIGSTOP

```
#define P_SIGSTOP 0
```

Signals for PennOS.

Definition at line 7 of file signal.h.

4.18.1.5 P_SIGTERM

```
#define P_SIGTERM 2
```

Definition at line 9 of file signal.h.

4.18.1.6 P_WIFEXITED

User-level macros for waitpid status.

Definition at line 22 of file signal.h.

4.18.1.7 P_WIFSIGNALED

Definition at line 24 of file signal.h.

4.18.1.8 P_WIFSTOPPED

Definition at line 23 of file signal.h.

4.18.1.9 STOPPED_BY_SIG

```
#define STOPPED_BY_SIG 21
```

Definition at line 15 of file signal.h.

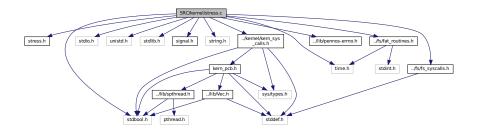
4.18.1.10 TERM_BY_SIG

```
#define TERM_BY_SIG 22
```

Definition at line 16 of file signal.h.

4.19 SRC/kernel/stress.c File Reference

```
#include "stress.h"
#include <stdbool.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include "../kernel/kern_sys_calls.h"
#include "../fs/fs_syscalls.h"
#include "../fs/fat_routines.h"
Include dependency graph for stress.c:
```



Functions

```
void * hang (void *arg)
void * nohang (void *arg)
void * recur (void *arg)
void * crash (void *arg)
```

4.19.1 Function Documentation

4.19.1.1 crash()

4.19.1.2 hang()

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

Definition at line 215 of file stress.c.

```
215 {
216    spawn(false);
217    s_exit();
218    return NULL;
219 }
```

4.19.1.3 nohang()

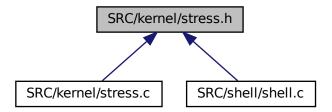
```
void* nohang (
     void * arg )
```

Definition at line 221 of file stress.c.

4.19.1.4 recur()

4.20 SRC/kernel/stress.h File Reference

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



Functions

```
void * hang (void *)
void * nohang (void *)
void * recur (void *)
void * crash (void *)
```

4.20.1 Function Documentation

4.20.1.1 crash()

238 }

4.20.1.2 hang()

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

Definition at line 215 of file stress.c.

```
215 {
216    spawn(false);
217    s_exit();
218    return NULL;
219 }
```

4.20.1.3 nohang()

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

Definition at line 221 of file stress.c.

```
221 {
222 spawn(true);
223 s_exit();
224 return NULL;
225 }
```

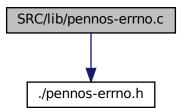
4.20.1.4 recur()

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

Definition at line 227 of file stress.c.

4.21 SRC/lib/pennos-errno.c File Reference

```
#include "./pennos-errno.h"
Include dependency graph for pennos-errno.c:
```



Variables

```
• int P_ERRNO = 0
```

4.21.1 Variable Documentation

4.21.1.1 P_ERRNO

```
int P\_ERRNO = 0
```

Definition at line 8 of file pennos-errno.c.

4.22 SRC/lib/pennos-errno.h File Reference

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



Macros

- #define P_ENOENT 1
- #define P_EBADF 2
- #define P_EPERM 3
- #define P_EINVAL 4
- #define P EEXIST 5
- #define P_EBUSY 6
- #define P_EFULL 7
- #define P_EFS_NOT_MOUNTED 8
- #define P EINTR 9
- #define P_ENULL 10
- #define P_EREAD 11
- #define P_ELSEEK 12
- #define P_EMAP 13
- #define P_EFUNC 14
- #define P_EOPEN 15
- #define P EMALLOC 16
- #define P_ESIGNAL 17
- #define P_EWRITE 18
- #define P_ECLOSE 19
- #define P_EPARSE 20
- #define P_ECOMMAND 21
- #define P_NEEDF 22
- #define P INITFAIL 23
- #define P_EUNKNOWN 99

Variables

• int P_ERRNO

4.22.1 Macro Definition Documentation

4.22.1.1 P_EBADF

```
#define P_EBADF 2
```

Definition at line 8 of file pennos-errno.h.

4.22.1.2 P_EBUSY

```
#define P_EBUSY 6
```

Definition at line 12 of file pennos-errno.h.

4.22.1.3 P_ECLOSE

```
#define P_ECLOSE 19
```

Definition at line 25 of file pennos-errno.h.

4.22.1.4 P_ECOMMAND

```
#define P_ECOMMAND 21
```

Definition at line 27 of file pennos-errno.h.

4.22.1.5 P_EEXIST

```
#define P_EEXIST 5
```

Definition at line 11 of file pennos-errno.h.

4.22.1.6 P_EFS_NOT_MOUNTED

```
#define P_EFS_NOT_MOUNTED 8
```

Definition at line 14 of file pennos-errno.h.

4.22.1.7 P_EFULL

```
#define P_EFULL 7
```

Definition at line 13 of file pennos-errno.h.

4.22.1.8 P_EFUNC

```
#define P_EFUNC 14
```

Definition at line 20 of file pennos-errno.h.

4.22.1.9 P_EINTR

```
#define P_EINTR 9
```

Definition at line 15 of file pennos-errno.h.

4.22.1.10 P EINVAL

```
#define P_EINVAL 4
```

Definition at line 10 of file pennos-errno.h.

4.22.1.11 P_ELSEEK

```
#define P_ELSEEK 12
```

Definition at line 18 of file pennos-errno.h.

4.22.1.12 P_EMALLOC

```
#define P_EMALLOC 16
```

Definition at line 22 of file pennos-errno.h.

4.22.1.13 P_EMAP

```
#define P_EMAP 13
```

Definition at line 19 of file pennos-errno.h.

4.22.1.14 P_ENOENT

```
#define P_ENOENT 1
```

Definition at line 7 of file pennos-errno.h.

4.22.1.15 P_ENULL

```
#define P_ENULL 10
```

Definition at line 16 of file pennos-errno.h.

4.22.1.16 P EOPEN

```
#define P_EOPEN 15
```

Definition at line 21 of file pennos-errno.h.

4.22.1.17 P_EPARSE

#define P_EPARSE 20

Definition at line 26 of file pennos-errno.h.

4.22.1.18 P_EPERM

#define P_EPERM 3

Definition at line 9 of file pennos-errno.h.

4.22.1.19 P_EREAD

#define P_EREAD 11

Definition at line 17 of file pennos-errno.h.

4.22.1.20 P_ESIGNAL

#define P_ESIGNAL 17

Definition at line 23 of file pennos-errno.h.

4.22.1.21 P_EUNKNOWN

#define P_EUNKNOWN 99

Definition at line 30 of file pennos-errno.h.

4.22.1.22 P EWRITE

#define P_EWRITE 18

Definition at line 24 of file pennos-errno.h.

4.22.1.23 P_INITFAIL

#define P_INITFAIL 23

Definition at line 29 of file pennos-errno.h.

4.22.1.24 P_NEEDF

```
#define P_NEEDF 22
```

Definition at line 28 of file pennos-errno.h.

4.22.2 Variable Documentation

4.22.2.1 P_ERRNO

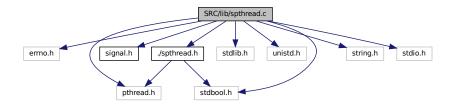
```
int P_ERRNO [extern]
```

Definition at line 8 of file pennos-errno.c.

4.23 SRC/lib/spthread.c File Reference

```
#include <errno.h>
#include <pthread.h>
#include <signal.h>
#include <stdbool.h>
#include <stdlib.h>
#include <unistd.h>
#include "./spthread.h"
#include <string.h>
#include <stdio.h>
```

Include dependency graph for spthread.c:



Classes

- struct spthread_fwd_args_st
- struct spthread_signal_args_st
- struct spthread_meta_st

Macros

- #define GNU SOURCE
- #define MILISEC_IN_NANO 100000
- #define SPTHREAD RUNNING STATE 0
- #define SPTHREAD SUSPENDED STATE 1
- #define SPTHREAD TERMINATED STATE 2
- #define SPTHREAD_SIG_SUSPEND -1
- #define SPTHREAD_SIG_CONTINUE -2

Typedefs

- typedef void *(* pthread_fn) (void *)
- typedef struct spthread_fwd_args_st spthread_fwd_args
- typedef struct spthread_signal_args_st spthread_signal_args
- typedef struct spthread_meta_st spthread_meta_t

Functions

- int spthread_create (spthread_t *thread, const pthread_attr_t *attr, pthread_fn start_routine, void *arg)
- int spthread_suspend (spthread_t thread)
- int spthread suspend self ()
- int spthread_continue (spthread_t thread)
- int spthread_cancel (spthread_t thread)
- bool spthread_self (spthread_t *thread)
- int spthread_join (spthread_t thread, void **retval)
- void spthread_exit (void *status)
- bool spthread_equal (spthread_t first, spthread_t second)
- int spthread_disable_interrupts_self ()
- int spthread_enable_interrupts_self ()

4.23.1 Macro Definition Documentation

4.23.1.1 _GNU_SOURCE

```
#define _GNU_SOURCE
```

Definition at line 1 of file spthread.c.

4.23.1.2 MILISEC_IN_NANO

```
#define MILISEC_IN_NANO 100000
```

Definition at line 12 of file spthread.c.

4.23.1.3 SPTHREAD_RUNNING_STATE

#define SPTHREAD_RUNNING_STATE 0

Definition at line 76 of file spthread.c.

4.23.1.4 SPTHREAD_SIG_CONTINUE

```
#define SPTHREAD_SIG_CONTINUE -2
```

Definition at line 85 of file spthread.c.

4.23.1.5 SPTHREAD_SIG_SUSPEND

#define SPTHREAD_SIG_SUSPEND -1

Definition at line 84 of file spthread.c.

4.23.1.6 SPTHREAD_SUSPENDED_STATE

#define SPTHREAD_SUSPENDED_STATE 1

Definition at line 77 of file spthread.c.

4.23.1.7 SPTHREAD_TERMINATED_STATE

#define SPTHREAD_TERMINATED_STATE 2

Definition at line 78 of file spthread.c.

4.23.2 Typedef Documentation

4.23.2.1 pthread_fn

typedef void*(* pthread_fn) (void *)

Definition at line 20 of file spthread.c.

4.23.2.2 spthread_fwd_args

```
typedef struct spthread_fwd_args_st spthread_fwd_args
```

4.23.2.3 spthread meta t

```
typedef struct spthread_meta_st spthread_meta_t
```

4.23.2.4 spthread_signal_args

```
{\tt typedef struct spthread\_signal\_args\_st spthread\_signal\_args}
```

4.23.3 Function Documentation

4.23.3.1 spthread_cancel()

return pthread_cancel(thread.thread);

4.23.3.2 spthread continue()

Definition at line 241 of file spthread.c.

```
241
        pthread_t pself = pthread_self();
242
243
        if (pthread_equal(pself, thread.thread) != 0) {
   // I am already runnning... so just return 0
   my_meta->state = SPTHREAD_RUNNING_STATE;
244
245
246
247
248
249
        spthread_signal_args args = (spthread_signal_args) {
   .signal = SPTHREAD_SIG_CONTINUE,
250
251
252
             .ack = 0,
253
254
       pthread_mutex_init(&args.shutup_mutex, NULL);
255
256
        int ret = pthread_sigqueue(thread.thread, SIGPTHD,
257
                                              (union sigval) {
258
                                                   .sival_ptr = &args,
```

```
259
                                     });
260
      if (ret != 0) {
261
        pthread_mutex_destroy(&args.shutup_mutex);
2.62
        \ensuremath{//} handles the case where the thread is already dead.
2.63
        return ret;
264
265
266
      // wait for our signal to be ack'd
267
268
      \ensuremath{//} setting up args to nanosleep
      const struct timespec t = (struct timespec) {
    .tv_nsec = MILISEC_IN_NANO,
269
270
271
272
273
      pthread_mutex_lock(&args.shutup_mutex);
      while (args.ack != 1) {
   // wait for a mili second
274
275
276
        pthread_mutex_unlock(&args.shutup_mutex);
277
278
        nanosleep(&t, NULL);
279
        // fprintf(stderr, "susp checking...\n");
280
        pthread_mutex_lock(&args.shutup_mutex);
2.81
282
283
        if (thread.meta->state == SPTHREAD_TERMINATED_STATE) {
         // child called exit, can break
284
285
286
        }
287
288
     pthread_mutex_unlock(&args.shutup_mutex);
289
      pthread_mutex_destroy(&args.shutup_mutex);
290
      return ret;
291 }
```

4.23.3.3 spthread_create()

spthread_t * thread,

const pthread_attr_t * attr,

int spthread_create (

143 144

145

146

147

148 149 150 free(child_meta);

free(fwd_args);

return EAGAIN;

pthread_t pthread;

```
pthread_fn start_routine,
                void * arg )
Definition at line 114 of file spthread.c.
117
      spthread_meta_t* child_meta = malloc(sizeof(spthread_meta_t));
118
119
      if (child_meta == NULL) {
120
       return EAGAIN;
121
122
      spthread_fwd_args* fwd_args = malloc(sizeof(spthread_fwd_args));
if (fwd_args == NULL) {
123
124
125
       free(child_meta);
        return EAGAIN;
126
127
128
      *fwd_args = (spthread_fwd_args){
          .actual_routine = start_routine,
129
          .actual_arg = arg,
130
          .setup_done = false,
131
132
          .child_meta = child_meta,
133
      };
134
      int ret = pthread_mutex_init(&(fwd_args->setup_mutex), NULL);
if (ret != 0) {
135
136
        free (child_meta);
137
138
        free(fwd_args);
139
        return EAGAIN;
140
141
142
      ret = pthread_cond_init(&(fwd_args->setup_cond), NULL);
```

pthread_mutex_destroy(&(fwd_args->setup_mutex));

```
151
      int result = pthread_create(&pthread, attr, spthread_start, fwd_args);
152
153
      pthread_mutex_lock(&(fwd_args->setup_mutex));
154
      while (fwd_args->setup_done == false) {
       pthread_cond_wait(&(fwd_args->setup_cond), &(fwd_args->setup_mutex));
155
156
157
     pthread_mutex_unlock(&(fwd_args->setup_mutex));
158
159
      pthread_cond_destroy(&(fwd_args->setup_cond));
160
      pthread_mutex_destroy(&(fwd_args->setup_mutex));
161
      free(fwd_args);
162
163
      *thread = (spthread_t) {
       .thread = pthread,
164
165
         .meta = child_meta,
166
167
168
     return result;
169 }
```

4.23.3.4 spthread_disable_interrupts_self()

```
int spthread_disable_interrupts_self ( )
Definition at line 326 of file spthread.c.
```

```
326
327
      sigset_t block_set;
328
      int res = sigemptyset(&block_set);
329
      if (res != 0) {
     return res;
}
330
331
     res = sigaddset(&block_set, SIGPTHD);
332
333
     if (res != 0) {
334
       return res;
335
336
     res = pthread_sigmask(SIG_BLOCK, &block_set, NULL);
     return res;
}
337
     if (res != 0) {
338
339
340
     return 0;
341 }
```

4.23.3.5 spthread_enable_interrupts_self()

```
int spthread_enable_interrupts_self ( )
```

Definition at line 345 of file spthread.c.

```
346
      sigset_t block_set;
      int res = sigemptyset(&block_set);
if (res != 0) {
347
348
     , res != 0)
return res;
}
349
350
351
      res = sigaddset(&block_set, SIGPTHD);
352
      if (res != 0) {
353
       return res;
354
355
      res = pthread_sigmask(SIG_UNBLOCK, &block_set, NULL);
356
      if (res != 0) {
357
       return res;
359
      return 0;
360 }
```

4.23.3.6 spthread_equal()

Definition at line 322 of file spthread.c.

```
322
323   return pthread_equal(first.thread, second.thread) && (first.meta == second.meta);
324 }
```

4.23.3.7 spthread_exit()

```
void spthread_exit (
     void * status )
```

Definition at line 315 of file spthread.c.

```
315 {
316    // necessary cleanup is registered
317    // in a cleanup routine
318    // that is pushed at start of an spthread
319    pthread_exit(status);
320 }
```

4.23.3.8 spthread_join()

Definition at line 308 of file spthread.c.

```
308
309  int res = pthread_join(thread.thread, retval);
310  pthread_mutex_destroy(&thread.meta->meta_mutex);
311  free(thread.meta);
312  return res;
313 }
```

4.23.3.9 spthread self()

Definition at line 297 of file spthread.c.

```
297
298    if (my_meta == NULL) {
299        return false;
300    }
301    *thread = (spthread_t) {
302        .thread = pthread_self(),
303        .meta = my_meta,
304    };
305    return true;
306 }
```

4.23.3.10 spthread_suspend()

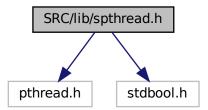
```
int spthread_suspend (
               spthread_t thread )
Definition at line 171 of file spthread.c.
171
172
      pthread_t pself = pthread_self();
173
174
      if (pthread_equal(pself, thread.thread) != 0) {
175
       return spthread_suspend_self();
      }
176
177
178
      spthread_signal_args args = (spthread_signal_args) {
179
          .signal = SPTHREAD_SIG_SUSPEND,
180
          .ack = 0,
181
182
      pthread_mutex_init(&args.shutup_mutex, NULL);
183
184
      int ret = pthread_sigqueue(thread.thread, SIGPTHD,
185
                                   (union sigval) {
186
                                       .sival_ptr = &args,
187
188
      if (ret != 0) {
189
       pthread_mutex_destroy(&args.shutup_mutex);
190
        \ensuremath{//} handles the case where the thread is already dead.
191
        return ret;
192
193
194
      // wait for our signal to be ack'd
195
      // setting up args to nanosleep
const struct timespec t = (struct timespec) {
   .tv_nsec = MILISEC_IN_NANO,
196
197
198
199
200
201
      nanosleep(&t, NULL);
202
203
      pthread_mutex_lock(&args.shutup_mutex);
204
      while (args.ack != 1) {
205
       // wait for a mili second
206
        pthread_mutex_unlock(&args.shutup_mutex);
207
208
209
        nanosleep(&t, NULL);
210
211
        // fprintf(stderr, "susp checking...\n");
212
        pthread_mutex_lock(&args.shutup_mutex);
213
214
        if (thread.meta->state == SPTHREAD_TERMINATED_STATE) {
          // child called exit, can break
215
216
          break:
217
        }
219
      pthread_mutex_unlock(&args.shutup_mutex);
220
221
      pthread_mutex_destroy(&args.shutup_mutex);
222
      return ret;
223 }
```

4.23.3.11 spthread_suspend_self()

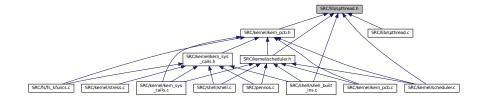
```
int spthread_suspend_self ( )
Definition at line 225 of file spthread.c.
225
226
      spthread_t self;
227
      bool am_sp = spthread_self(&self);
228
      if (!am_sp) {
229
       return ESRCH;
     }
230
231
232
     my_meta->state = SPTHREAD_SUSPENDED_STATE;
233
234
235
       sigsuspend(&my_meta->suspend_set);
     } while (my_meta->state == SPTHREAD_SUSPENDED_STATE);
236
237
238
     return 0;
239 }
```

4.24 SRC/lib/spthread.h File Reference

#include <pthread.h>
#include <stdbool.h>
Include dependency graph for spthread.h:



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



Classes

• struct spthread_st

Macros

• #define SIGPTHD SIGUSR1

Typedefs

- typedef struct spthread_meta_st spthread_meta_t
- typedef struct spthread_st spthread_t

Functions

- int spthread_create (spthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)
- int spthread_suspend (spthread_t thread)
- int spthread suspend self ()
- int spthread_continue (spthread_t thread)
- int spthread_cancel (spthread_t thread)
- bool spthread_self (spthread_t *thread)
- int spthread_join (spthread_t thread, void **retval)
- void spthread_exit (void *status)
- bool spthread_equal (spthread_t first, spthread_t second)
- int spthread_disable_interrupts_self ()
- int spthread_enable_interrupts_self ()

4.24.1 Macro Definition Documentation

4.24.1.1 SIGPTHD

#define SIGPTHD SIGUSR1

Definition at line 19 of file spthread.h.

4.24.2 Typedef Documentation

4.24.2.1 spthread_meta_t

typedef struct spthread_meta_st spthread_meta_t

Definition at line 1 of file spthread.h.

4.24.2.2 spthread_t

typedef struct spthread_st spthread_t

4.24.3 Function Documentation

4.24.3.1 spthread_cancel()

4.24.3.2 spthread_continue()

Definition at line 241 of file spthread.c.

```
241
242
      pthread_t pself = pthread_self();
243
244
      if (pthread_equal(pself, thread.thread) != 0) {
       // I am already running... so just return 0
my_meta->state = SPTHREAD_RUNNING_STATE;
245
246
247
        return 0:
248
249
250
      spthread_signal_args args = (spthread_signal_args) {
251
          .signal = SPTHREAD_SIG_CONTINUE,
252
          .ack = 0,
253
254
      pthread_mutex_init(&args.shutup_mutex, NULL);
255
256
      int ret = pthread_sigqueue(thread.thread, SIGPTHD,
257
                                   (union sigval) {
258
                                        .sival_ptr = &args,
259
                                   });
260
      if (ret != 0) {
261
       pthread_mutex_destroy(&args.shutup_mutex);
262
        // handles the case where the thread is already dead.
263
        return ret;
264
265
266
      // wait for our signal to be ack'd
267
268
      // setting up args to nanosleep
269
      const struct timespec t = (struct timespec) {
270
          .tv_nsec = MILISEC_IN_NANO,
271
272
273
      pthread_mutex_lock(&args.shutup_mutex);
      while (args.ack != 1) {
   // wait for a mili second
274
275
276
        pthread_mutex_unlock(&args.shutup_mutex);
277
278
        nanosleep(&t, NULL);
279
        // fprintf(stderr, "susp checking...\n");
280
281
        pthread_mutex_lock(&args.shutup_mutex);
282
283
        if (thread.meta->state == SPTHREAD_TERMINATED_STATE) {
284
          // child called exit, can break
285
          break;
286
287
288
      pthread_mutex_unlock(&args.shutup_mutex);
289
      pthread_mutex_destroy(&args.shutup_mutex);
290
      return ret;
291 }
```

4.24.3.3 spthread_create()

```
int spthread_create (
               spthread_t * thread,
                const pthread_attr_t * attr,
                void *(*)(void *) start_routine,
                void * arg )
Definition at line 114 of file spthread.c.
117
118
      spthread_meta_t* child_meta = malloc(sizeof(spthread_meta_t));
      if (child_meta == NULL) {
119
120
        return EAGAIN;
121
122
      spthread_fwd_args* fwd_args = malloc(sizeof(spthread_fwd_args));
123
      if (fwd_args == NULL) {
124
       free(child_meta);
return EAGAIN;
125
126
127
128
      *fwd_args = (spthread_fwd_args){
          .actual_routine = start_routine,
129
130
          .actual_arg = arg,
.setup_done = false,
131
          .child_meta = child_meta,
132
133
      };
134
135
      int ret = pthread_mutex_init(&(fwd_args->setup_mutex), NULL);
if (ret != 0) {
136
137
        free (child meta):
138
        free(fwd_args);
139
        return EAGAIN;
140
141
      ret = pthread_cond_init(&(fwd_args->setup_cond), NULL);
if (ret != 0) {
142
143
144
        free(child_meta);
145
        pthread_mutex_destroy(&(fwd_args->setup_mutex));
146
        free(fwd_args);
147
        return EAGAIN;
148
149
150
      pthread_t pthread;
151
      int result = pthread_create(&pthread, attr, spthread_start, fwd_args);
152
153
      pthread_mutex_lock(&(fwd_args->setup_mutex));
      while (fwd_args->setup_done == false) {
   pthread_cond_wait(&(fwd_args->setup_cond), &(fwd_args->setup_mutex));
154
155
156
157
      pthread_mutex_unlock(&(fwd_args->setup_mutex));
158
159
      pthread_cond_destroy(&(fwd_args->setup_cond));
160
      pthread_mutex_destroy(&(fwd_args->setup_mutex));
161
      free(fwd_args);
162
163
      *thread = (spthread_t) {
164
       .thread = pthread,
165
          .meta = child_meta,
166
      };
167
168
      return result;
169 }
```

4.24.3.4 spthread disable interrupts self()

331

```
332
     res = sigaddset(&block_set, SIGPTHD);
333
     if (res != 0) {
     return res;
}
334
335
     res = pthread_sigmask(SIG_BLOCK, &block_set, NULL);
336
     - (res != 0)
   return res;
}
337
     if (res != 0) {
338
339
340 return 0;
341 }
```

4.24.3.5 spthread_enable_interrupts_self()

```
int spthread_enable_interrupts_self ( )
```

Definition at line 345 of file spthread.c.

```
345
     sigset_t block_set;
     int res = sigemptyset(&block_set);
if (res != 0) {
347
348
349
      return res;
350
351
     res = sigaddset(&block_set, SIGPTHD);
352 if (res != 0) {
     return res;
}
353
354
     res = pthread_sigmask(SIG_UNBLOCK, &block_set, NULL);
355
356 if (res != 0) {
359 return 0;
360 }
```

4.24.3.6 spthread_equal()

Definition at line 322 of file spthread.c.

```
322
323   return pthread_equal(first.thread, second.thread) && (first.meta == second.meta);
324 }
```

4.24.3.7 spthread_exit()

```
void spthread_exit (
     void * status )
```

Definition at line 315 of file spthread.c.

```
315 {
316    // necessary cleanup is registered
317    // in a cleanup routine
318    // that is pushed at start of an spthread
319    pthread_exit(status);
320 }
```

4.24.3.8 spthread_join()

4.24.3.9 spthread_self()

Definition at line 297 of file spthread.c.

```
297
298    if (my_meta == NULL) {
299        return false;
300    }
301    *thread = (spthread_t) {
302        .thread = pthread_self(),
303        .meta = my_meta,
304    };
305    return true;
306 }
```

4.24.3.10 spthread suspend()

Definition at line 171 of file spthread.c.

```
171
172
      pthread_t pself = pthread_self();
173
174
      if (pthread_equal(pself, thread.thread) != 0) {
175
       return spthread_suspend_self();
176
177
178
      spthread_signal_args args = (spthread_signal_args) {
         .signal = SPTHREAD_SIG_SUSPEND,
180
          .ack = 0,
181
182
     pthread_mutex_init(&args.shutup_mutex, NULL);
183
      int ret = pthread_sigqueue(thread.thread, SIGPTHD,
184
185
                                  (union sigval) {
186
                                      .sival_ptr = &args,
187
188
      if (ret != 0) {
189
      pthread_mutex_destroy(&args.shutup_mutex);
190
        \ensuremath{//} handles the case where the thread is already dead.
191
        return ret;
192
193
194
     // wait for our signal to be ack'd
195
196
     // setting up args to nanosleep
197
      const struct timespec t = (struct timespec) {
198
         .tv_nsec = MILISEC_IN_NANO,
```

```
199
       } ;
200
201
       nanosleep(&t, NULL);
202
      pthread_mutex_lock(&args.shutup_mutex);
while (args.ack != 1) {
    // wait for a mili second
203
204
205
206
        pthread_mutex_unlock(&args.shutup_mutex);
207
208
209
        nanosleep(&t, NULL);
210
         // fprintf(stderr, "susp checking...\n");
211
         pthread_mutex_lock(&args.shutup_mutex);
212
213
         if (thread.meta->state == SPTHREAD_TERMINATED_STATE) {
  // child called exit, can break
214
215
216
           break;
217
218
219
      pthread_mutex_unlock(&args.shutup_mutex);
220
221 pthread_mutex_destroy(&args.shutup_mutex);
222
       return ret;
223 }
```

4.24.3.11 spthread_suspend_self()

```
int spthread_suspend_self ( )
```

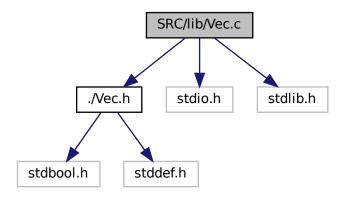
Definition at line 225 of file spthread.c.

```
226
      spthread_t self;
      bool am_sp = spthread_self(&self);
if (!am_sp) {
227
228
     ,.am_sp) {
  return ESRCH;
}
229
230
231
232
      my_meta->state = SPTHREAD_SUSPENDED_STATE;
233
234
235
       sigsuspend(&my_meta->suspend_set);
      } while (my_meta->state == SPTHREAD_SUSPENDED_STATE);
236
237
238
     return 0;
239 }
```

4.25 SRC/lib/Vec.c File Reference

```
#include "./Vec.h"
#include <stdio.h>
#include <stdlib.h>
```

Include dependency graph for Vec.c:



Functions

- Vec vec_new (size_t initial_capacity, ptr_dtor_fn ele_dtor_fn)
- void vec_destroy (Vec *self)
- void vec_clear (Vec *self)
- void vec_resize (Vec *self, size_t new_capacity)
- void vec_erase (Vec *self, size_t index)
- void vec erase no deletor (Vec *self, size t index)
- void vec_insert (Vec *self, size_t index, ptr_t new_ele)
- bool vec_pop_back (Vec *self)
- void vec_push_back (Vec *self, ptr_t new_ele)
- void vec_set (Vec *self, size_t index, ptr_t new_ele)
- ptr_t vec_get (Vec *self, size_t index)

4.25.1 Function Documentation

4.25.1.1 vec_clear()

Erases all elements from the container. After this, the length of the vector is zero. Capacity of the vector is unchanged.

Parameters

self	a pointer to the vector we want to clear.

Precondition

Assumes self points to a valid vector.

Postcondition

The removed elements are destructed (cleaned up).

Definition at line 34 of file Vec.c.

4.25.1.2 vec_destroy()

Destruct the vector. All elements are destructed and storage is deallocated. Must set capacity and length to zero. Data is set to NULL.

Parameters

```
self | a pointer to the vector we want to destruct.
```

Precondition

Assumes self points to a valid vector.

Postcondition

The removed elements are destructed (cleaned up) and data storage deallocated.

Definition at line 17 of file Vec.c.

4.25.1.3 vec_erase()

Erases an element at the specified valid location in the container

Parameters

self	a pointer to the vector we want to erase from.	
inde	the index of the element we want to erase at. Elements after this index are "shifted" down one position.	

Precondition

Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

Definition at line 63 of file Vec.c.

4.25.1.4 vec_erase_no_deletor()

Erases an element at the specified location in the container, except it does not call the element deletor function on the element.

Parameters

self	a pointer to the vector we want to erase from
index	the index of the element we want to erase at

Definition at line 79 of file Vec.c.

```
79
80    if (index >= self->length) {
81        perror("vec_erase: index >= vec length");
82    }
83
84    for (unsigned int i = index; i < self->length - 1; i++) {
        self->data[i] = self->data[i + 1];
86    }
87
88    self->length--;
89 }
```

4.25.1.5 vec_get()

Gets the specified element of the Vec

Parameters

self	a pointer to the vector who's element we want to get.
index	the index of the element to get.

Returns

the element at the specified index.

Precondition

Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

Definition at line 152 of file Vec.c.

```
152
153    if (index >= self->length) {
154        perror("vec_get: index greater than length");
155    }
156    return self->data[index];
157 }
```

4.25.1.6 vec_insert()

Inserts an element at the specified location in the container

Parameters

self	a pointer to the vector we want to insert into.
index	the index of the element we want to insert at. Elements at this index and after it are "shifted" up one position. If index is equal to the length, then we insert at the end of the vector.
new_ele	the value we want to insert

Precondition

Assumes self points to a valid vector. If the index is > self->length then this function will call perror

Postcondition

If after the operation the new length is greater than the old capacity then a reallocation takes place and all elements are copied over. Capacity is doubled. Any pointers to elements prior to this reallocation are invalidated.

Definition at line 91 of file Vec.c.

```
if (index > self->length) {
      perror("vec_insert: index > vec length");
93
    }
94
95
   if (index == self->length) { // Insertion at end = Adding at end
96
      vec_push_back(self, new_ele);
   } else { // Inserting not at the end
     // Vector is full
99
      if (self->length == self->capacity) {
100
         vec_resize(self, self->capacity * 2);
101
102
      // Insertion + Displacement
for (unsigned int i = self->length; i > index; i--) {
103
104
105
         self->data[i] = self->data[i - 1];
106
107
       self->data[index] = new_ele;
108
109
       self->length++;
111 }
```

4.25.1.7 vec_new()

Creates a new empty Vec(tor) with the specified initial_capacity and specified function to clean up elements in the vector.

Parameters

initial_capacity	the initial capacity of the newly created vector, non negative
ele_dtor_fn	a function pointer to a function that takes in a ptr_t (a vector element) and cleans it up. This is commonly just free but custom functions can be passed in. NULL can also be passed in to specify that there is no cleanup function that needs to be called on each element.

Returns

a newly created vector with specified capacity, 0 length and the specified element destructor (cleanup) function.

Postcondition

if memory allocation fails, the function will perror

Definition at line 5 of file Vec.c.

```
5
6  Vec vector;
7  vector.capacity = initial_capacity;
8  vector.data = malloc(sizeof(void*) * initial_capacity);
9  if (vector.data == NULL) {
10    perror("vec_new: malloc failed");
11  }
12  vector.ele_dtor_fn = ele_dtor_fn;
13  vector.length = 0;
14  return vector;
15 }
```

4.25.1.8 vec_pop_back()

```
bool vec_pop_back ( \label{eq:vec_pop_back} \mbox{Vec} \ * \ self \mbox{)}
```

Removes and destroys the last element of the Vec

Parameters

```
self a pointer to the vector we are popping.
```

Returns

true iff an element was removed.

Precondition

Assumes self points to a valid vector.

Postcondition

The capacity of self stays the same. The removed element is destructed (cleaned up) as specified by the dtor_fn provided in vec_new.

Definition at line 113 of file Vec.c.

4.25.1.9 vec_push_back()

Appends the given element to the end of the Vec

Parameters

self	a pointer to the vector we are pushing onto
new_ele	the value we want to add to the end of the container

Precondition

Assumes self points to a valid vector.

Postcondition

If a resize is needed and it fails, then this function will call perror

If after the operation the new length is greater than the old capacity then a reallocation takes place and all elements are copied over. Capacity is doubled. If initial capacity is zero, it is resized to capacity 1. Any pointers to elements prior to this reallocation are invalidated.

Definition at line 125 of file Vec.c.

```
126
      if (self->capacity == self->length) {
127
      if (self->capacity == 0) {
128
         vec_resize(self, 1);
129
       } else {
130
         vec_resize(self, self->capacity * 2);
131
132
133
135 perror("vec_push_back: resize failed");
136 }
137
138
     // The array is 0 indexed
     self->data[self->length++] = new_ele;
140 }
```

4.25.1.10 vec_resize()

Resizes the container to a new specified capacity. Does nothing if new_capacity <= self->length

Parameters

self	a pointer to the vector we want to resize.
new_capacity	the new capacity of the vector.

Precondition

Assumes self points to a valid vector.

Postcondition

If a resize takes place, then a reallocation takes place and all elements are copied over. Any pointers to elements prior to this reallocation are invalidated.

The removed elements are destructed (cleaned up).

Definition at line 44 of file Vec.c.

```
44
45    if (new_capacity * sizeof(void*) < new_capacity) {
46        perror("vec_resize: new capacity too large");</pre>
```

```
47  }
48  if (new_capacity > self->length) {
49    self->capacity = new_capacity;
50    ptr_t* new_data = malloc(sizeof(void*) * self->capacity);
51
52    // Copy over old elements
53    for (int i = 0; i < self->length; i++) {
54        new_data[i] = self->data[i];
55    }
56
57    free(self->data);
58
59    self->data = new_data;
60  }
61 }
```

4.25.1.11 vec_set()

Sets the specified element of the Vec to the specified value

Parameters

self	a pointer to the vector who's element we want to set.
index	the index of the element to set.
new_ele	the value we want to set the element at that index to

Returns

the element at the specified index.

Precondition

Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

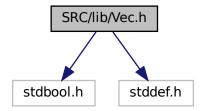
Definition at line 142 of file Vec.c.

```
142
143     if (index >= self->length) {
144         perror("vec_set: idx >= len");
145     }
146     if (self->ele_dtor_fn) {
147         self->ele_dtor_fn(self->data[index]);
148     }
149     self->data[index] = new_ele;
150 }
```

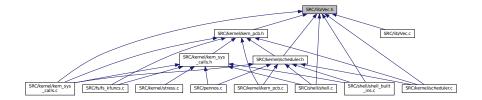
4.26 SRC/lib/Vec.h File Reference

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

Include dependency graph for Vec.h:



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



Classes

• struct vec_st

Macros

- #define vec_capacity(vec) ((vec)->capacity)
- #define vec_len(vec) ((vec)->length)
- #define vec_is_empty(vec) ((vec)->length == 0)

Typedefs

- typedef void * ptr_t
- typedef void(* ptr_dtor_fn) (ptr_t)
- typedef struct vec_st Vec

Functions

- Vec vec_new (size_t initial_capacity, ptr_dtor_fn ele_dtor_fn)
- ptr_t vec_get (Vec *self, size_t index)
- void vec_set (Vec *self, size_t index, ptr_t new_ele)
- void vec_push_back (Vec *self, ptr_t new_ele)
- bool vec_pop_back (Vec *self)
- void vec_insert (Vec *self, size_t index, ptr_t new_ele)
- void vec_erase (Vec *self, size_t index)
- void vec erase no deletor (Vec *self, size t index)
- void vec_resize (Vec *self, size_t new_capacity)
- void vec_clear (Vec *self)
- void vec_destroy (Vec *self)

4.26.1 Macro Definition Documentation

4.26.1.1 vec_capacity

Returns the current capacity of the Vec Written as a function-like macro

Parameters

vec,a pointer to the vector we want to grab the capacity of.

Definition at line 40 of file Vec.h.

4.26.1.2 vec_is_empty

```
#define vec_is_empty( vec \ ) \ \ ((vec) -> length == 0)
```

Checks if the Vec is empty written as a function-like macro

Parameters

vec,a pointer to the vector we want to check emptiness of.

Definition at line 54 of file Vec.h.

4.26.1.3 vec_len

```
#define vec_len( vec \ ) \ \ \mbox{((vec)->length)}
```

Returns the current length of the Vec written as a function-like macro

Parameters

vec,a pointer to the vector we want to grab the len of.

Definition at line 47 of file Vec.h.

4.26.2 Typedef Documentation

4.26.2.1 ptr_dtor_fn

```
typedef void(* ptr_dtor_fn) (ptr_t)
```

Definition at line 8 of file Vec.h.

4.26.2.2 ptr_t

```
typedef void* ptr_t
```

Definition at line 7 of file Vec.h.

4.26.2.3 Vec

```
typedef struct vec_st Vec
```

4.26.3 Function Documentation

4.26.3.1 vec_clear()

Erases all elements from the container. After this, the length of the vector is zero. Capacity of the vector is unchanged.

Parameters

self | a pointer to the vector we want to clear.

Precondition

Assumes self points to a valid vector.

Postcondition

The removed elements are destructed (cleaned up).

Definition at line 34 of file Vec.c.

4.26.3.2 vec_destroy()

Destruct the vector. All elements are destructed and storage is deallocated. Must set capacity and length to zero. Data is set to NULL.

Parameters

self | a pointer to the vector we want to destruct.

Precondition

Assumes self points to a valid vector.

Postcondition

The removed elements are destructed (cleaned up) and data storage deallocated.

Definition at line 17 of file Vec.c.

4.26.3.3 vec_erase()

Erases an element at the specified valid location in the container

Parameters

self	a pointer to the vector we want to erase from.	
inde	the index of the element we want to erase at. Elements after this index are "shifted" down one position.	

Precondition

Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

Definition at line 63 of file Vec.c.

4.26.3.4 vec_erase_no_deletor()

Erases an element at the specified location in the container, except it does not call the element deletor function on the element.

Parameters

self	a pointer to the vector we want to erase from	
index	the index of the element we want to erase at	

Definition at line 79 of file Vec.c.

```
79
80    if (index >= self->length) {
81        perror("vec_erase: index >= vec length");
82    }
83
84    for (unsigned int i = index; i < self->length - 1; i++) {
        self->data[i] = self->data[i + 1];
86    }
87
88    self->length--;
89 }
```

4.26.3.5 vec_get()

Gets the specified element of the Vec

Parameters

self	a pointer to the vector who's element we want to get.
index	the index of the element to get.

Returns

the element at the specified index.

Precondition

Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

Definition at line 152 of file Vec.c.

```
152
153   if (index >= self->length) {
154     perror("vec_get: index greater than length");
155   }
156   return self->data[index];
157 }
```

4.26.3.6 vec_insert()

Inserts an element at the specified location in the container

Parameters

self	a pointer to the vector we want to insert into.
index	the index of the element we want to insert at. Elements at this index and after it are "shifted" up one position. If index is equal to the length, then we insert at the end of the vector.
new_ele	the value we want to insert

Precondition

Assumes self points to a valid vector. If the index is > self->length then this function will call perror

Postcondition

If after the operation the new length is greater than the old capacity then a reallocation takes place and all elements are copied over. Capacity is doubled. Any pointers to elements prior to this reallocation are invalidated.

Definition at line 91 of file Vec.c.

```
if (index > self->length) {
      perror("vec_insert: index > vec length");
93
    }
94
95
   if (index == self->length) { // Insertion at end = Adding at end
96
      vec_push_back(self, new_ele);
   } else { // Inserting not at the end
     // Vector is full
99
       if (self->length == self->capacity) {
100
         vec_resize(self, self->capacity * 2);
101
102
      // Insertion + Displacement
for (unsigned int i = self->length; i > index; i--) {
103
104
105
         self->data[i] = self->data[i - 1];
106
107
       self->data[index] = new_ele;
108
109
       self->length++;
111 }
```

4.26.3.7 vec_new()

Creates a new empty Vec(tor) with the specified initial_capacity and specified function to clean up elements in the vector.

Parameters

initial_capacity	the initial capacity of the newly created vector, non negative
ele_dtor_fn	a function pointer to a function that takes in a ptr_t (a vector element) and cleans it up. This is commonly just free but custom functions can be passed in. NULL can also be passed in to specify that there is no cleanup function that needs to be called on each element.

Returns

a newly created vector with specified capacity, 0 length and the specified element destructor (cleanup) function.

Postcondition

if memory allocation fails, the function will perror

Definition at line 5 of file Vec.c.

```
5
6  Vec vector;
7  vector.capacity = initial_capacity;
8  vector.data = malloc(sizeof(void*) * initial_capacity);
9  if (vector.data == NULL) {
10    perror("vec_new: malloc failed");
11  }
12  vector.ele_dtor_fn = ele_dtor_fn;
13  vector.length = 0;
14  return vector;
15 }
```

4.26.3.8 vec_pop_back()

```
bool vec_pop_back ( \label{eq:vec_pop_back} \mbox{Vec} \ * \ self \mbox{)}
```

Removes and destroys the last element of the Vec

Parameters

```
self a pointer to the vector we are popping.
```

Returns

true iff an element was removed.

Precondition

Assumes self points to a valid vector.

Postcondition

The capacity of self stays the same. The removed element is destructed (cleaned up) as specified by the dtor_fn provided in vec_new.

Definition at line 113 of file Vec.c.

4.26.3.9 vec_push_back()

Appends the given element to the end of the Vec

Parameters

self	a pointer to the vector we are pushing onto
new_ele	the value we want to add to the end of the container

Precondition

Assumes self points to a valid vector.

Postcondition

If a resize is needed and it fails, then this function will call perror

If after the operation the new length is greater than the old capacity then a reallocation takes place and all elements are copied over. Capacity is doubled. If initial capacity is zero, it is resized to capacity 1. Any pointers to elements prior to this reallocation are invalidated.

Definition at line 125 of file Vec.c.

```
126
      if (self->capacity == self->length) {
127
      if (self->capacity == 0) {
128
         vec_resize(self, 1);
129
       } else {
130
         vec_resize(self, self->capacity * 2);
131
132
133
135 perror("vec_push_back: resize failed");
136 }
137
138
     // The array is 0 indexed
     self->data[self->length++] = new_ele;
140 }
```

4.26.3.10 vec_resize()

Resizes the container to a new specified capacity. Does nothing if new_capacity <= self->length

Parameters

self	a pointer to the vector we want to resize.
new_capacity	the new capacity of the vector.

Precondition

Assumes self points to a valid vector.

Postcondition

If a resize takes place, then a reallocation takes place and all elements are copied over. Any pointers to elements prior to this reallocation are invalidated.

The removed elements are destructed (cleaned up).

Definition at line 44 of file Vec.c.

```
44
45    if (new_capacity * sizeof(void*) < new_capacity) {
46        perror("vec_resize: new capacity too large");</pre>
```

```
47  }
48  if (new_capacity > self->length) {
49    self->capacity = new_capacity;
50    ptr_t* new_data = malloc(sizeof(void*) * self->capacity);
51
52    // Copy over old elements
53    for (int i = 0; i < self->length; i++) {
54        new_data[i] = self->data[i];
55    }
56
57    free(self->data);
58
59    self->data = new_data;
60  }
61 }
```

4.26.3.11 vec_set()

Sets the specified element of the Vec to the specified value

Parameters

self	a pointer to the vector who's element we want to set.
index	the index of the element to set.
new_ele	the value we want to set the element at that index to

Returns

the element at the specified index.

Precondition

Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

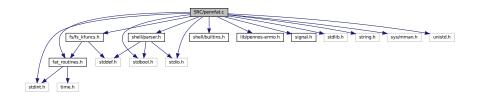
Definition at line 142 of file Vec.c.

```
142
143    if (index >= self->length) {
144         perror("vec_set: idx >= len");
145    }
146    if (self->ele_dtor_fn) {
147         self->ele_dtor_fn(self->data[index]);
148    }
149    self->data[index] = new_ele;
150 }
```

4.27 SRC/pennfat.c File Reference

```
#include "shell/parser.h"
#include "shell/builtins.h"
#include "lib/pennos-errno.h"
#include "fs/fs_kfuncs.h"
```

```
#include "fs/fat_routines.h"
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stdbool.h>
Include dependency graph for pennfat.c:
```



Macros

• #define PROMPT "pennfat# "

Functions

• int main (int argc, char *argv[])

4.27.1 Macro Definition Documentation

4.27.1.1 PROMPT

```
#define PROMPT "pennfat# "
```

Definition at line 16 of file pennfat.c.

4.27.2 Function Documentation

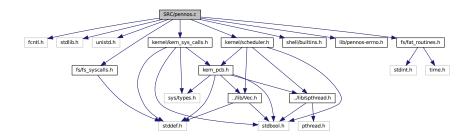
4.27.2.1 main()

```
int main (
                 int argc,
                 char * argv[] )
Definition at line 24 of file pennfat.c.
25
        // register signal handlers
26
        struct sigaction sa;
2.7
        sa.sa_handler = signal_handler;
        sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
28
29
30
        // set up handler for SIGINT (ctrl-c)
32
        if (sigaction(SIGINT, &sa, NULL) == -1) {
33
             P_ERRNO = P_ESIGNAL;
            u_perror("Error setting up SIGINT handler");
34
3.5
            return EXIT_FAILURE;
36
37
        // set up handler for SIGTSTP (ctrl-z)
39
        if (sigaction(SIGTSTP, &sa, NULL) == -1) {
40
             P_ERRNO = P_ESIGNAL;
             u_perror("Error setting up SIGTSTP handler");
41
            return EXIT FAILURE:
42
43
44
45
        char input_buffer[1024];
46
        while (true) {
47
48
             // print prompt
49
             if (k_write(STDOUT_FILENO, PROMPT, strlen(PROMPT)) < 0) {</pre>
                 P_ERRNO = P_EWRITE;
51
                 u_perror("prompt write error");
52
                 break;
53
            }
54
55
             // read user input
            int bytes_read = k_read(STDIN_FILENO, input_buffer, sizeof(input_buffer) - 1);
56
            // check for EOF (ctrl-D)
if (bytes_read <= 0) {</pre>
58
59
                 k_write(STDOUT_FILENO, "\n", 1);
60
61
                 break:
62
             }
64
             \ensuremath{//} remove trailing newline if present
6.5
            if (bytes_read > 0 && input_buffer[bytes_read - 1] == '\n') {
                 input_buffer[bytes_read - 1] = ' \setminus 0';
66
67
68
             // parse command and check error
70
             struct parsed_command *parsed_command = NULL;
71
             int parse_result = parse_command(input_buffer, &parsed_command);
            if (parse_result != 0) {
   if (parse_result == -1) {
     P_ERRNO = P_EINVAL;
}
72
73
74
75
                      u_perror("Error parsing command");
76
                 } else {
77
                     print_parser_errcode(stderr, parse_result);
78
79
                 continue:
             }
80
82
             // skip empty commands
83
             if (parsed_command->num_commands == 0) {
84
                 free (parsed_command);
85
                 continue;
86
             }
87
             // extract command and arguments
89
            char **args = parsed_command->commands[0];
90
91
             // execute command
             if (strcmp(args[0], "mkfs") == 0) {
   if (args[1] == NULL || args[2] == NULL || args[3] == NULL) {
        P_ERRNO = P_EINVAL;
92
93
95
                      u_perror("mkfs");
96
                 } else {
                      int blocks_in_fat = atoi(args[2]);
97
                      int block_size = atoi(args[3]);
if (mkfs(args[1], blocks_in_fat, block_size) != 0) {
98
99
100
                            u_perror("mkfs");
```

```
101
102
             else\ if\ (strcmp(args[0], "mount") == 0) {
103
                 if (args[1] == NULL) {
   P_ERRNO = P_EINVAL;
104
                     u_perror("mount");
106
107
                 } else {
108
                     if (mount(args[1]) != 0) {
109
                         u_perror("mount");
110
                 }
111
             } else if (strcmp(args[0], "unmount") == 0) {
112
                if (unmount() != 0) {
    u_perror("unmount");
113
114
115
116
             else if (strcmp(args[0], "ls") == 0) {
117
                 ls(args);
            } else if (strcmp(args[0], "touch") == 0) {
118
119
                touch (args);
             } else if (strcmp(args[0], "cat") == 0) {
121
                 cat (args);
122
             } else if (strcmp(args[0], "chmod") == 0) {
123
                chmod(args);
            } else if (strcmp(args[0], "mv") == 0) {
124
125
                 mv(args);
126
            } else if (strcmp(args[0], "rm") == 0) {
127
                 rm(args);
128
             } else if (strcmp(args[0], "cp") == 0) {
129
                 cp (args);
             } else if (strcmp(args[0], "cmpctdir") == 0) { // extra credit
130
131
                cmpctdir(args);
132
             } else {
133
                 P_ERRNO = P_ECOMMAND;
134
                 u_perror("shell");
135
136
137
             free(parsed_command);
138
139
        return EXIT_SUCCESS;
140 }
```

4.28 SRC/pennos.c File Reference

```
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include "fs/fs_syscalls.h"
#include "kernel/kern_sys_calls.h"
#include "kernel/scheduler.h"
#include "shell/builtins.h"
#include "lib/pennos-errno.h"
#include "fs/fat_routines.h"
Include dependency graph for pennos.c:
```



Functions

• int main (int argc, char *argv[])

Variables

- · int tick_counter
- int log fd

4.28.1 Function Documentation

4.28.1.1 main()

```
int main (
                  int argc,
                  char * argv[] )
Definition at line 14 of file pennos.c.
14
      // mount the filesystem
15
16
      if (argc < 2) {</pre>
17
      P_ERRNO = P_NEEDF;
        u_perror("need a pennfat file to mount");
18
19
       return -1;
    } else {
20
      if (mount(argv[1]) == -1) {
  u_perror("mount failed");
21
23
          return -1;
24
      }
    }
25
2.6
     // get the log fd
if (argc >= 3) {
2.7
28
       log_fd = open(argv[2], O_RDWR | O_CREAT | O_TRUNC, 0644);
30
31
        log_fd = open("log/log", O_RDWR | O_CREAT | O_TRUNC, 0644);
32
33
     // initialize scheduler architecture and init process
34
35
     initialize_scheduler_queues();
36
    pid_t init_pid = s_spawn_init();
if (init_pid == -1) {
  P_ERRNO = P_INITFAIL;
  u_perror("init spawn failed");
37
38
39
40
        return -1;
42
43
44
     scheduler();
4.5
     // cleanup
46
    s_cleanup_init_process();
free_scheduler_queues();
49
     unmount();
50 close(log_fd);
51 }
```

4.28.2 Variable Documentation

4.28.2.1 log_fd

```
int log_fd [extern]
```

Definition at line 36 of file scheduler.c.

4.28.2.2 tick_counter

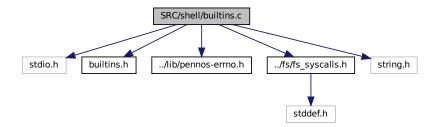
```
int tick_counter [extern]
```

Definition at line 35 of file scheduler.c.

4.29 SRC/shell/builtins.c File Reference

```
#include <stdio.h>
#include "builtins.h"
#include "../lib/pennos-errno.h"
#include "../fs/fs_syscalls.h"
#include <string.h>
```

Include dependency graph for builtins.c:



Functions

void u_perror (const char *msg)

4.29.1 Function Documentation

4.29.1.1 u_perror()

Creates a user-level error message similar to perror.

Parameters

msg A string representing the error message from the shell.

Definition at line 12 of file builtins.c.

```
12
                                         {
13
        char buffer[256];
14
        const char *error_msg;
1.5
16
        switch (P ERRNO) {
            case P_ENOENT:
17
                  error_msg = "file does not exist";
18
19
20
             case P_EBADF:
                  error_msg = "bad file descriptor";
2.1
22
                 break;
             case P_EPERM:
23
24
                  error_msg = "operation not permitted";
25
                 break;
26
             case P_EINVAL:
                error_msg = "invalid arg";
2.7
            break;
case P_EEXIST:
28
29
                error_msg = "file already exists";
30
                  break;
32
             case P_EBUSY:
                  error_msg = "file is busy or open";
33
34
                  break;
             case P_EFULL:
35
36
                  error_msg = "no space left on device";
                  break;
38
             case P_EINTR:
39
                  error_msg = "interrupted system call";
             break; case P_ENULL:
40
41
                  error_msg = "NULL returned unexpectedly";
42
43
                  break;
44
             case P_EUNKNOWN:
                  error_msg = "unknown error";
45
             break;
case P_EREAD:
46
47
                  error_msg = "interrupted read call";
48
49
                 break;
             case P_ELSEEK:
                error_msg = "interrupted lseek call";
51
                 break;
52
             case P_EMAP:
5.3
                error_msg = "interrupted mmap/munmap call";
54
55
                  break;
             case P_EFUNC:
56
57
                  error_msg = "interrupted system call";
58
                  break;
             case P_EOPEN:
59
                  error_msg = "interrupted open call";
60
61
                  break:
             case P_EMALLOC:
63
                  error_msg = "error when trying to malloc";
64
             case P_EFS_NOT_MOUNTED:
    error_msg = "file system not mounted yet";
65
66
                 break;
             case P_ESIGNAL:
69
                  error_msg = "error with signal handling";
70
                 break;
             case P_EWRITE:
71
                  error_msg = "interrupted write call";
72
73
                 break;
74
             case P_ECLOSE:
75
                error_msg = "interrupted close call";
76
77
             case P_EPARSE:
                error_msg = "error when trying to parse a command";
78
79
                  break:
             case P_ECOMMAND:
80
                  error_msg = "command not found";
81
82
8.3
             case P_NEEDF:
                  error_msg = "no file provided to mount";
84
85
                  break;
86
             default:
                  error_msg = "Unknown error";
88
89
90
        \label{eq:snprintf} $$\operatorname{snprintf}(\operatorname{buffer}, \operatorname{sizeof}(\operatorname{buffer}), \ "\$s: \$s\n", \operatorname{msg}, \operatorname{error\_msg}); $$\operatorname{s\_write}(\operatorname{STDERR\_FILENO}, \operatorname{buffer}, \operatorname{strlen}(\operatorname{buffer})); $$
91
92
```

4.30 SRC/shell/builtins.h File Reference

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



Functions

void u_perror (const char *msg)

4.30.1 Function Documentation

4.30.1.1 u_perror()

Creates a user-level error message similar to perror.

Parameters

msg A string representing the error message from the shell.

Definition at line 12 of file builtins.c.

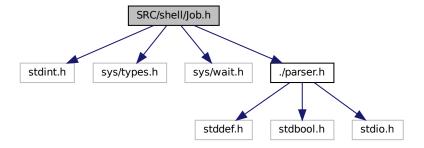
```
13
       char buffer[256];
14
       const char *error_msg;
15
      switch (P_ERRNO) {
16
         case P_ENOENT:
               error_msg = "file does not exist";
18
19
20
           case P_EBADF:
               error_msg = "bad file descriptor";
21
           break;
case P_EPERM:
22
23
               error_msg = "operation not permitted";
25
               break;
           case P_EINVAL:
              error_msg = "invalid arg";
27
2.8
          break;
case P_EEXIST:
29
30
              error_msg = "file already exists";
32
           case P_EBUSY:
               error_msg = "file is busy or open";
33
           break;
case P_EFULL:
34
35
36
               error_msg = "no space left on device";
               break;
           case P_EINTR:
38
39
               error_msg = "interrupted system call";
```

```
break;
           case P_ENULL:
               error_msg = "NULL returned unexpectedly";
42
           break;
case P_EUNKNOWN:
43
44
             error_msg = "unknown error";
break;
45
47
           case P_EREAD:
             error_msg = "interrupted read call";
48
               break;
49
           case P_ELSEEK:
50
              error_msg = "interrupted lseek call";
51
               break;
           case P_EMAP:
54
              error_msg = "interrupted mmap/munmap call";
           break;
case P_EFUNC:
55
56
               error_msg = "interrupted system call";
57
58
               break;
           case P_EOPEN:
60
               error_msg = "interrupted open call";
          break;
case P_EMALLOC:
61
62
               error_msg = "error when trying to malloc";
6.3
64
               break;
           case P_EFS_NOT_MOUNTED:
               error_msg = "file system not mounted yet";
67
              break;
           case P_ESIGNAL:
68
             error_msg = "error with signal handling";
69
70
               break:
          case P_EWRITE:
             error_msg = "interrupted write call";
72
73
74
75
          case P_ECLOSE:
             error_msg = "interrupted close call";
76
               break:
           case P_EPARSE:
78
              error_msg = "error when trying to parse a command";
79
80
           case P_ECOMMAND:
               error_msg = "command not found";
81
82
               break;
           case P_NEEDF:
83
             error_msg = "no file provided to mount";
85
86
           default:
               error_msg = "Unknown error";
87
88
               break:
89
91
       snprintf(buffer, sizeof(buffer), "%s: %s\n", msg, error_msg);
92
       s_write(STDERR_FILENO, buffer, strlen(buffer));
93 1
```

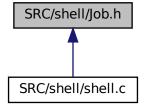
4.31 SRC/shell/Job.h File Reference

```
#include <stdint.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "./parser.h"
```

Include dependency graph for Job.h:



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



Classes

• struct job_st

Typedefs

- typedef uint64_t jid_t
- typedef struct job_st job

Enumerations

enum job_state_t { RUNNING , STOPPED , FINISHED }

4.31.1 Typedef Documentation

4.31.1.1 jid_t

```
typedef uint64_t jid_t
```

Definition at line 10 of file Job.h.

4.31.1.2 job

```
typedef struct job_st job
```

4.31.2 Enumeration Type Documentation

4.31.2.1 job_state_t

```
enum job_state_t
```

Enumerator

RUNNING	
STOPPED	
FINISHED	

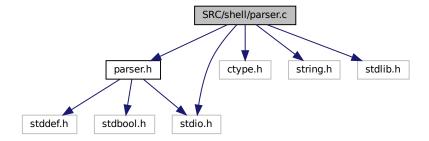
```
Definition at line 13 of file Job.h.
```

```
13 { RUNNING, STOPPED, FINISHED } job_state_t;
```

4.32 SRC/shell/parser.c File Reference

```
#include "parser.h"
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
```

Include dependency graph for parser.c:



Macros

• #define JUMP_OUT(code) do {ret_code = code; goto PROCESS_ERROR;} while (0)

Functions

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

4.32.1 Macro Definition Documentation

4.32.1.1 JUMP_OUT

4.32.2 Function Documentation

4.32.2.1 parse_command()

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

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

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

Otherwise, no struct parsed_command is allocated and *result is unchanged. If a system call error (-1) is returned, the caller can use errno(3) or perror(3) to gain more information about the error. layout of memory for struct parsed_command bool is_background; bool is_file_append;

const char *stdin file; const char *stdout file;

size_t num_commands;

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

below are hidden in memory **

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

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

Definition at line 16 of file parser.c.

```
17 #define JUMP_OUT(code) do {ret_code = code; goto PROCESS_ERROR;} while (0)
19
       int ret code = -1;
21
       const char *start = cmd_line;
2.2
       const char *end = cmd_line + strlen(cmd_line);
23
24
       for (const char *cur = start; cur < end; ++cur)</pre>
            if (*cur == '#') {
                 // all subsequent characters following '#'
                 \ensuremath{//} shall be discarded as a comment.
27
2.8
                 end = cur;
29
                break:
30
            }
31
        // trimming leading and trailing whitespaces
       while (start < end && isspace(*start)) ++start;
while (start < end && isspace(end[-1])) --end;</pre>
33
34
35
       struct parsed_command *pcmd = calloc(1, sizeof(struct parsed_command));
36
        if (pcmd == NULL) return -1;
       if (start == end) goto PROCESS_SUCCESS; // empty line, fast pass
38
39
40
        // If a command is terminated by the control operator ampersand ( ^{\prime} & ^{\prime} ),
41
       \ensuremath{//} the shell shall execute the command in background.
42
       if (end[-1] == '&') {
            pcmd->is_background = true;
43
             -end;
```

```
45
       }
46
47
        // first pass, check token
48
       int total_strings = 0; // number of total arguments
49
            bool has_token_last = false, has_file_input = false, has_file_output = false;
50
51
            const char *skipped;
            for (const char *cur = start; cur < end; skip_space(&cur, end))</pre>
52
53
                switch (cur[0]) {
54
                     case '&':
                         JUMP_OUT(UNEXPECTED_AMPERSAND); // does not expect anymore ampersand
55
                     case '<':
56
                         // if already had pipeline or had file input, error
                         if (pcmd->num_commands > 0 || has_file_input) JUMP_OUT(UNEXPECTED_FILE_INPUT);
58
59
60
                         ++cur; // skip '<'
61
                         skip_space(&cur, end);
62
63
                         // test if we indeed have a filename following '<'
                         skipped = cur;
                         skip_word(&skipped, end);
65
66
                         if (skipped <= cur) JUMP_OUT(EXPECT_INPUT_FILENAME);</pre>
67
                         \ensuremath{//} fast-forward to the end of the filename
68
69
                         cur = skipped;
                         has_file_input = true;
70
71
                         break;
                     case '>':
72
                         // if already had file output, error
if (has_file_output) JUMP_OUT(UNEXPECTED_FILE_OUTPUT);
if (cur + 1 < end && cur[1] == '>') { // dealing with 'w' append
73
74
75
76
                             pcmd->is_file_append = true;
77
                              ++cur;
78
79
                         ++cur; // skip '>'
80
                         skip_space(&cur, end);
81
82
                          // test filename, as the case above
                         skipped = cur;
84
85
                         skip_word(&skipped, end);
                         if (skipped <= cur) JUMP_OUT(EXPECT_OUTPUT_FILENAME);</pre>
86
87
88
                         // fast-forward to the end of the filename
                         cur = skipped;
                         has_file_output = true;
90
91
                     break; case '|':
92
                         \ensuremath{//} if already had file output but encourter a pipeline, it should
93
                         // rather be a file output error instead of a pipeline one. if (has_file_output) JUMP_OUT(UNEXPECTED_FILE_OUTPUT);
94
95
                          // if no tokens between two pipelines (or before the first one)
96
97
                          // should throw a pipeline error
98
                          if (!has_token_last) JUMP_OUT(UNEXPECTED_PIPELINE);
99
                         has_token_last = false;
                          ++pcmd->num_commands;
100
                          ++cur; // skip '|'
101
102
                          break;
103
                      default:
104
                          has_token_last = true;
105
                           ++total_strings;
                          skip_word(&cur, end); // skip that argument
106
107
                 }
108
109
             if (total_strings == 0) {
                 \ensuremath{//} if there are no arguments but has ampersand or file input/output
110
                  // then we have an error
111
                 if (pcmd->is_background || has_file_input || has_file_output)
112
                      JUMP_OUT (EXPECT_COMMANDS);
113
114
                 // otherwise it's an empty line
115
                 goto PROCESS_SUCCESS;
116
117
             // handle edge case where the command ends with a pipeline
118
             // (not supporting line continuation)
119
120
             if (!has_token_last) JUMP_OUT(UNEXPECTED_PIPELINE);
121
122
         ++pcmd->num_commands;
123
146
        const size_t start_of_array = offsetof(struct parsed_command, commands) +pcmd->num_commands *
147
       sizeof(char **);
148
        const size_t start_of_str = start_of_array + (pcmd->num_commands + total_strings) * sizeof(char *);
149
         const size_t slen = end - start;
150
         char *const new_buf = realloc(pcmd, start_of_str + slen + 1);
151
         if (new_buf == NULL) goto PROCESS_ERROR;
152
```

```
153
        pcmd = (struct parsed_command *) new_buf;
154
155
         // copy string to the new place
        char *const new_start = memcpy(new_buf + start_of_str, start, slen);
156
157
158
         // second pass, put stuff in
         // no need to check for error anymore
159
160
         size_t cur_cmd = 0;
161
        char **argv_ptr = (char **) (new_buf + start_of_array);
162
         pcmd->commands[cur_cmd] = argv_ptr;
163
         for (const char *cur = start; cur < end; skip_space(&cur, end)) {
164
165
             switch (cur[0]) {
166
                 case '<':
167
                     ++cur;
                      skip_space(&cur, end);
// store input file name into `stdin_file`
168
169
                      pcmd->stdin_file = new_start + (cur - start);
170
171
                      skip_word(&cur, end);
172
                      // at end of the input file name
173
                      new_start[cur - start] = ' \setminus 0';
174
                 break;
case '>':
175
                     if (pcmd->is_file_append) ++cur; // skip another '>'
176
177
                      ++cur;
178
                      skip_space(&cur, end);
// store output file name into `stdout_file`
179
180
                      pcmd->stdout_file = new_start + (cur - start);
181
                      skip_word(&cur, end);
                      \ensuremath{//} at end of the output file name
182
183
                      new_start[cur - start] = ' \setminus 0';
184
                      break;
185
186
                     // null-terminate the current argv
187
                      *(argv_ptr++) = NULL;
                      \ensuremath{//} store the next argv head
188
                      pcmd->commands[++cur_cmd] = argv_ptr;
189
190
                      ++cur;
191
                      break;
192
                 default:
                     // at start of the argument string
// store it into the arguments array
193
194
                      *(argv_ptr++) = new_start + (cur - start);
195
                      skip_word(&cur, end);
196
                      // at end of the argument string
197
198
                      new_start[cur - start] = '\0';
199
             }
200
        // null-terminate the last argv
201
        *argv_ptr = NULL;
202
203
204 PROCESS_SUCCESS:
205
        *result = pcmd;
206
         return 0:
207 PROCESS ERROR:
208
        free (pcmd);
209
         return ret_code;
210 }
```

4.32.2.2 print_parsed_command()

```
void print_parsed_command (
               const struct parsed_command *const cmd )
Definition at line 214 of file parser.c.
214
215
        for (size_t i = 0; i < cmd->num_commands; ++i) {
216
            for (char **arguments = cmd->commands[i]; *arguments != NULL; ++arguments)
217
                 printf("%s ", *arguments);
218
            if (i == 0 && cmd->stdin_file != NULL)
    printf("< %s ", cmd->stdin_file);
219
220
221
            if (i == cmd->num_commands - 1) {
223
                 if (cmd->stdout_file != NULL)
224
                     printf(cmd->is_file_append ? "» %s " : "> %s ", cmd->stdout_file);
225
            } else printf("| ");
226
227
        puts(cmd->is_background ? "&" : "");
228 }
```

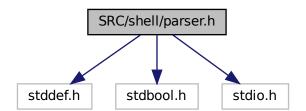
4.32.2.3 print_parser_errcode()

```
void print_parser_errcode (
                FILE * output,
                 int err_code )
Definition at line 230 of file parser.c.
2.31
       switch (err_code) {
  case UNEXPECTED_FILE_INPUT:
232
233
          fprintf(output, "UNEXPECTED INPUT REDIRECTION TO A FILE\n");
234
235
        case UNEXPECTED_FILE_OUTPUT:
         fprintf(output, "UNEXPECTED OUTPUT REDIRECTION TO A FILE\n");
236
        break;
case UNEXPECTED_PIPELINE:
237
238
         fprintf(output, "UNEXPECTED PIPE\n");
239
           break;
241
        case UNEXPECTED_AMPERSAND:
        fprintf(output, "UNEXPECTED AMPERESAND\n");
break;
case EXPECT_INPUT_FILENAME:
    fprintf(output, "COULD NOT FINE FILENAME FOR INPUT REDIRECTION \"<\"\n");
    break;</pre>
242
243
244
245
247
        case EXPECT_OUTPUT_FILENAME:
248
         fprintf(output, "COULD NOT FIND FILENAME FOR OUTPUT REDIRECTION \"<\"\n");
249
        case EXPECT COMMANDS:
250
251
         fprintf(output, "COULD NOT FIND ANY COMMANDS OR ARGS\n");
252
253
254
           break;
255 }
256 }
```

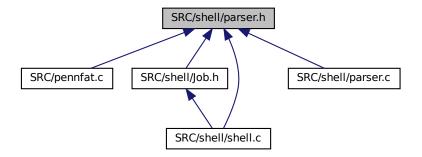
4.33 SRC/shell/parser.h File Reference

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

Include dependency graph for parser.h:



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



Classes

· struct parsed_command

Macros

- #define UNEXPECTED_FILE_INPUT 1
- #define UNEXPECTED_FILE_OUTPUT 2
- #define UNEXPECTED_PIPELINE 3
- #define UNEXPECTED AMPERSAND 4
- #define EXPECT INPUT FILENAME 5
- #define EXPECT_OUTPUT_FILENAME 6
- #define EXPECT_COMMANDS 7

Functions

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

4.33.1 Macro Definition Documentation

4.33.1.1 EXPECT_COMMANDS

#define EXPECT_COMMANDS 7

Definition at line 30 of file parser.h.

4.33.1.2 EXPECT_INPUT_FILENAME

#define EXPECT_INPUT_FILENAME 5

Definition at line 24 of file parser.h.

4.33.1.3 EXPECT_OUTPUT_FILENAME

#define EXPECT_OUTPUT_FILENAME 6

Definition at line 27 of file parser.h.

4.33.1.4 UNEXPECTED_AMPERSAND

#define UNEXPECTED_AMPERSAND 4

Definition at line 21 of file parser.h.

4.33.1.5 UNEXPECTED_FILE_INPUT

#define UNEXPECTED_FILE_INPUT 1

Definition at line 12 of file parser.h.

4.33.1.6 UNEXPECTED FILE OUTPUT

#define UNEXPECTED_FILE_OUTPUT 2

Definition at line 15 of file parser.h.

4.33.1.7 UNEXPECTED_PIPELINE

#define UNEXPECTED_PIPELINE 3

Definition at line 18 of file parser.h.

4.33.2 Function Documentation

4.33.2.1 parse_command()

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 successfully -1: parser encountered a system call error 1-7: parser specific error, see error type above

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

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

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

Otherwise, no struct parsed_command is allocated and *result is unchanged. If a system call error (-1) is returned, the caller can use errno(3) or perror(3) to gain more information about the error. layout of memory for struct parsed_command bool is_background; bool is_file_append;

const char *stdin_file; const char *stdout_file;

size_t num_commands;

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

below are hidden in memory **

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

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

Definition at line 16 of file parser.c.

```
17 #define JUMP_OUT(code) do {ret_code = code; goto PROCESS_ERROR;} while (0)
19
       int ret_code = -1;
20
       const char *start = cmd_line;
21
       const char *end = cmd line + strlen(cmd line);
22
23
       for (const char *cur = start; cur < end; ++cur)</pre>
25
           if (*cur == '#') {
                // all subsequent characters following '#'
26
                \ensuremath{//} shall be discarded as a comment.
27
28
                end = cur;
                break;
30
32
       \ensuremath{//} trimming leading and trailing whitespaces
3.3
       while (start < end && isspace(*start)) ++start;</pre>
       while (start < end && isspace(end[-1])) --end;</pre>
34
35
       struct parsed_command *pcmd = calloc(1, sizeof(struct parsed_command));
```

```
if (pcmd == NULL) return -1;
       if (start == end) goto PROCESS_SUCCESS; // empty line, fast pass
38
39
40
       // If a command is terminated by the control operator ampersand ( ^\prime\,\epsilon^\prime ),
       // the shell shall execute the command in background. if (end[-1] == '&') {
41
42
           pcmd->is_background = true;
43
44
             -end;
45
       }
46
       // first pass, check token
47
48
       int total_strings = 0; // number of total arguments
49
           bool has_token_last = false, has_file_input = false, has_file_output = false;
50
51
            const char *skipped;
52
            for (const char *cur = start; cur < end; skip_space(&cur, end))</pre>
53
                switch (cur[0]) {
                    case '&':
54
                       JUMP_OUT(UNEXPECTED_AMPERSAND); // does not expect anymore ampersand
55
56
                    case '<':
                       // if already had pipeline or had file input, error
58
                        if (pcmd->num_commands > 0 || has_file_input) JUMP_OUT(UNEXPECTED_FILE_INPUT);
59
                        ++cur: // skip '<'
60
                        skip_space(&cur, end);
61
62
                        // test if we indeed have a filename following '<'
63
64
                        skipped = cur;
6.5
                        skip_word(&skipped, end);
                        if (skipped <= cur) JUMP_OUT(EXPECT_INPUT_FILENAME);</pre>
66
67
68
                        // fast-forward to the end of the filename
                        cur = skipped;
69
70
                        has_file_input = true;
                    break;
case '>':
71
72
                        // if already had file output, error
73
74
                        if (has_file_output) JUMP_OUT(UNEXPECTED_FILE_OUTPUT);
75
                        if (cur + 1 < end && cur[1] == '>') { // dealing with '>' append
76
                            pcmd->is_file_append = true;
77
                             ++cur;
78
                        }
79
80
                        ++cur; // skip '>'
81
                        skip_space(&cur, end);
82
83
                        // test filename, as the case above
84
                        skipped = cur;
                        skip_word(&skipped, end);
85
                        if (skipped <= cur) JUMP_OUT(EXPECT_OUTPUT_FILENAME);</pre>
86
88
                        // fast-forward to the end of the filename
29
                        cur = skipped;
90
                        has_file_output = true;
91
                        break;
                    case '|':
92
                        // if already had file output but encourter a pipeline, it should
                        // rather be a file output error instead of a pipeline one.
94
95
                        if (has_file_output) JUMP_OUT(UNEXPECTED_FILE_OUTPUT);
96
                        // if no tokens between two pipelines (or before the first one)
                        \ensuremath{//} should throw a pipeline error
97
98
                        if (!has_token_last) JUMP_OUT(UNEXPECTED_PIPELINE);
99
                        has_token_last = false;
                         ++pcmd->num_commands;
100
101
                         ++cur; // skip '|
102
                         break;
103
                     default:
104
                         has token last = true;
105
                         ++total strings:
106
                         skip_word(&cur, end); // skip that argument
107
108
109
            if (total_strings == 0) {
                 // if there are no arguments but has ampersand or file input/output
110
                 // then we have an error
111
112
                 if (pcmd->is_background || has_file_input || has_file_output)
113
                     JUMP_OUT (EXPECT_COMMANDS);
114
                 // otherwise it's an empty line
115
                 goto PROCESS_SUCCESS;
            }
116
117
118
             // handle edge case where the command ends with a pipeline
             // (not supporting line continuation)
119
120
             if (!has_token_last) JUMP_OUT(UNEXPECTED_PIPELINE);
121
122
        ++pcmd->num_commands;
123
```

```
146
         const size_t start_of_array = offsetof(struct parsed_command, commands) +pcmd->num_commands *
147
        sizeof(char **);
        const size_t start_of_str = start_of_array + (pcmd->num_commands + total_strings) * sizeof(char *);
148
        const size_t slen = end - start;
149
150
151
        char *const new_buf = realloc(pcmd, start_of_str + slen + 1);
152
         if (new_buf == NULL) goto PROCESS_ERROR;
153
        pcmd = (struct parsed_command *) new_buf;
154
155
         // copy string to the new place
        char *const new_start = memcpy(new_buf + start_of_str, start, slen);
156
157
158
         // second pass, put stuff in
159
         // no need to check for error anymore
160
         size_t cur_cmd = 0;
        char **argv_ptr = (char **) (new_buf + start_of_array);
161
162
163
        pcmd->commands[cur_cmd] = argv_ptr;
164
         for (const char *cur = start; cur < end; skip_space(&cur, end)) {</pre>
165
             switch (cur[0]) {
166
                  case '<':
                     ++cur;
167
                      skip_space(&cur, end);
// store input file name into `stdin_file`
168
169
170
                      pcmd->stdin_file = new_start + (cur - start);
171
                      skip_word(&cur, end);
172
                      \ensuremath{//} at end of the input file name
173
                      new_start[cur - start] = ' \setminus 0';
174
                      break:
                  case '>':
175
176
                      if (pcmd->is_file_append) ++cur; // skip another '>'
177
                      ++cur;
178
                      skip_space(&cur, end);
                      // store output file name into 'stdout_file'
pcmd->stdout_file = new_start + (cur - start);
179
180
                      skip_word(&cur, end);
181
                      // at end of the output file name
182
183
                      new_start[cur - start] = ' \setminus 0';
184
                  break; case '|':
185
                      // null-terminate the current argv
186
                      \star (argv_ptr++) = NULL;
187
188
                      // store the next argv head
                      pcmd->commands[++cur_cmd] = argv_ptr;
189
190
                       ++cur;
191
                      break;
192
                  default:
                      // at start of the argument string
// store it into the arguments array
193
194
                      *(argv_ptr++) = new_start + (cur - start);
195
196
                      skip_word(&cur, end);
197
                      \ensuremath{//} at end of the argument string
198
                      new_start[cur - start] = ' \setminus 0';
             }
199
200
        // null-terminate the last argv
201
202
        *argv_ptr = NULL;
203
204 PROCESS SUCCESS:
       *result = pcmd;
2.0.5
206
        return 0;
207 PROCESS_ERROR:
208
        free (pcmd);
209
         return ret_code;
210 }
```

4.33.2.2 print_parsed_command()

4.33.2.3 print_parser_errcode()

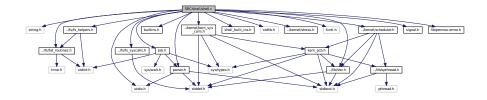
```
void print_parser_errcode (
               FILE * output,
               int err_code )
Definition at line 230 of file parser.c.
230
      switch (err_code) {
       case UNEXPECTED_FILE_INPUT:
233
          fprintf(output, "UNEXPECTED INPUT REDIRECTION TO A FILE\n");
234
        case UNEXPECTED_FILE_OUTPUT:
235
        fprintf (output, "UNEXPECTED OUTPUT REDIRECTION TO A FILE\n");
236
237
          break;
       case UNEXPECTED_PIPELINE:
        fprintf(output, "UNEXPECTED PIPE\n");
239
240
       case UNEXPECTED_AMPERSAND:
  fprintf(output, "UNEXPECTED AMPERESAND\n");
2.41
242
243
          break:
244
       case EXPECT_INPUT_FILENAME:
        fprintf(output, "COULD NOT FINE FILENAME FOR INPUT REDIRECTION \"<\"n");
245
246
        case EXPECT_OUTPUT_FILENAME:
    fprintf(output, "COULD NOT FIND FILENAME FOR OUTPUT REDIRECTION \"<\"\n");</pre>
247
248
249
          break;
        case EXPECT_COMMANDS:
        fprintf(output, "COULD NOT FIND ANY COMMANDS OR ARGS\n");
251
252
253
        default:
2.54
          break;
255
      }
256 }
```

4.34 SRC/shell/shell.c File Reference

```
#include <string.h>
#include "../fs/fat_routines.h"
#include "../fs/fs_syscalls.h"
#include "../fs/fs_helpers.h"
#include "../kernel/kern_sys_calls.h"
#include "builtins.h"
#include "parser.h"
#include "shell_built_ins.h"
#include "stdlib.h"
#include "../kernel/stress.h"
#include <fcntl.h>
#include "../kernel/scheduler.h"
#include "../lib/Vec.h"
#include "Job.h"
#include "signal.h"
#include "lib/pennos-errno.h"
```

#include "stdio.h"

Include dependency graph for shell.c:



Macros

- #define PROMPT "\$ "
- #define MAX BUFFER SIZE 4096
- #define MAX_LINE_BUFFER_SIZE 128

Functions

- void shell_sigint_handler (int sig)
- void shell_sigstp_handler (int sig)
- · void setup_terminal_signal_handlers (void)
- void free_job_ptr (void *ptr)
- void fill buffer until full or newline (int fd, char *buffer)

Helper function that fills a buffer with characters read from a given file descriptor until the buffer is full (rare and impractical case), a newline is encountered, or EOF is reached.

pid_t u_execute_command (struct parsed_command *cmd)

Helper function that will execute a given command so long as it's one of the built-ins. Notably, its output and input are determined by the spawning script process.

void * u_read_and_execute_script (void *arg)

Helper function that reads a script file line by line, parses each line as a command, and executes it.

pid_t execute_command (struct parsed_command *cmd)

Helper function to execute a parsed command from the shell. In particular, it spawns a child process to execute the command if the built-in should run as a separate process. Otherwise, it just calls the subroutine directly.

void * shell (void *)

The main shell function that runs the shell. This is run via an s_spawn call from init's process. It prompts the user for builtins and scripts to run.

Variables

- · pid_t current_fg_pid
- Vec job_list
- jid_t next_job_id = 1
- int script_fd = -1
- int input fd script = -1
- int output_fd_script = -1

4.34.1 Macro Definition Documentation

4.34.1.1 MAX_BUFFER_SIZE

```
#define MAX_BUFFER_SIZE 4096
```

Definition at line 29 of file shell.c.

4.34.1.2 MAX_LINE_BUFFER_SIZE

```
#define MAX_LINE_BUFFER_SIZE 128
```

Definition at line 30 of file shell.c.

4.34.1.3 PROMPT

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

Definition at line 26 of file shell.c.

4.34.2 Function Documentation

4.34.2.1 execute command()

Helper function to execute a parsed command from the shell. In particular, it spawns a child process to execute the command if the built-in should run as a separate process. Otherwise, it just calls the subroutine directly.

Parameters

```
cmd the parsed command to execute, assumed non-null
```

Returns

the created child id on successful spawn, 0 on successful subroutine call, -1 when nothing was called

Definition at line 257 of file shell.c.

```
257
258
259 // setup fds
260 int input_fd = STDIN_FILENO; // standard fds
261 int output_fd = STDOUT_FILENO;
262
```

```
263
        if (cmd->stdin_file != NULL) {
          input_fd = s_open(cmd->stdin_file, F_READ);
if (input_fd < 0) {</pre>
264
265
             input_fd = STDIN_FILENO; // reset to default
266
2.67
268
        }
269
270
        if (cmd->is_file_append) {
271
          output_fd = s_open(cmd->stdout_file, F_APPEND);
272
273
          output_fd = s_open(cmd->stdout_file, F_WRITE);
274
275
        if (output fd < 0) {
           output_fd = STDOUT_FILENO; // reset to default
276
277
278
        // check for independently scheduled processes
279
        if (strcmp(cmd->commands[0][0], "cat") == 0) {
   return s_spawn(u_cat, cmd->commands[0], input_fd, output_fd);
280
281
        } else if (strcmp(cmd->commands[0][0], "sleep") == 0) {
          return s_spawn(u_sleep, cmd->commands[0], input_fd, output_fd);
283
284
        } else if (strcmp(cmd->commands[0][0], "busy") == 0) {
       return s_spawn(u_busy, cmd->commands[0][0], "ousy") == 0) {
   return s_spawn(u_busy, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "echo") == 0) {
   return s_spawn(u_echo, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "ls") == 0) {
285
286
287
        return s_spawn(u_ls, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "touch") == 0) {
289
290
        return s_spawn(u_touch, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "mv") == 0) {
291
292
          else if (strcmp(cmd->commands[0], in, ...,
return s_spawn(u_mv, cmd->commands[0], input_fd, output_fd);
clos if (strcmp(cmd->commands[0][0], "cp") == 0) {
293
294
        } else if (strcmp(cmd->commands[0][0],
        return s_spawn(u_op, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "rm") == 0) {
295
296
        return s_spawn(u_rm, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "chmod") == 0) {
297
298
        return s_spawn(u_chmod, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "ps") == 0) {
299
300
        return s_spawn(u_ps, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "kill") == 0) {
301
302
        return s_spawn(u_kill, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "zombify") == 0) {
303
304
        return s_spawn(u_zombify, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "orphanify") == 0) {
305
306
        return s_spawn(u_orphanify, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "hang") == 0) {
308
        return s_spawn(hang, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "nohang") == 0) {
  return s_spawn(nohang, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "recur") == 0) {
309
310
311
312
           return s_spawn(recur, cmd->commands[0], input_fd, output_fd);
313
314
        } else if (strcmp(cmd->commands[0][0], "crash") == 0) {
315
           return s_spawn(crash, cmd->commands[0], input_fd, output_fd);
316
317
318
        // check for sub-routines
        if (strcmp(cmd->commands[0][0], "nice") == 0) {
319
           u_nice(cmd->commands[0]);
320
321
           return 0;
        } else if (strcmp(cmd->commands[0][0], "nice_pid") == 0) {
322
323
           u nice pid(cmd->commands[0]);
324
           return 0;
325
        } else if (strcmp(cmd->commands[0][0], "man") == 0) {
           u_man(cmd->commands[0]);
326
327
           return 0;
328
        } else if (strcmp(cmd->commands[0][0], "bg") == 0) {
329
          u_bg(cmd->commands[0]);
330
           return 0:
        } else if (strcmp(cmd->commands[0][0], "fq") == 0) {
331
332
           u_fg(cmd->commands[0]);
333
           return 0;
        } else if (strcmp(cmd->commands[0][0], "jobs") == 0) {
334
           u_jobs(cmd->commands[0]);
335
336
           return 0;
        } else if (strcmp(cmd->commands[0][0], "logout") == 0) {
337
338
           u_logout (cmd->commands[0]);
339
           return 0;
340
341
342
        // otherwise, try to run command as a script
        int script_fd_open = s_open(cmd->commands[0][0], F_READ);
343
        if (script_fd_open < 0) { // if not a file, just move on</pre>
344
345
346
347
        if (has_executable_permission(script_fd_open) != 1) {
348
           s_close(script_fd_open);
349
           return -1:
```

```
350
      } else {
351
         script_fd = script_fd_open; // update global
         input_fd_script = input_fd;
output_fd_script = output_fd;
352
353
354
355
         char* script_argv[] = {cmd->commands[0][0], NULL};
356
        pid_t wait_on = s_spawn(u_read_and_execute_script, script_argv, input_fd,
357
                    output_fd);
358
         int status;
         s_waitpid(wait_on, &status, false); // wait for script to finish
359
         script_fd = -1; // reset global
input_fd_script = STDIN_FILENO;
output_fd_script = STDOUT_FILENO;
360
361
362
363
         s_close(script_fd_open);
364
         return 0;
365
366
      return -1; // no matches case
367
```

4.34.2.2 fill_buffer_until_full_or_newline()

Helper function that fills a buffer with characters read from a given file descriptor until the buffer is full (rare and impractical case), a newline is encountered, or EOF is reached.

Parameters

fd	the file descriptor to read from, assumed to be open
buffer	the buffer to fill with characters

```
Definition at line 113 of file shell.c.
```

```
113
       int i = 0;
114
115
       char currChar;
116
       while (i < MAX_LINE_BUFFER_SIZE - 1) {</pre>
        int bytes_read = s_read(fd, &currChar, 1);
if (bytes_read <= 0 || currChar == '\n') { // EOF or newline cases</pre>
117
118
119
            break:
120
121
          buffer[i] = currChar;
122
123
      buffer[i] = ' \setminus 0'; // Null-terminate the string, replaces \setminus n
124
125 }
```

4.34.2.3 free_job_ptr()

```
void free_job_ptr ( \mbox{void} \ * \ ptr \ )
```

Definition at line 94 of file shell.c.

```
94
95    job* job_ptr = (job*)ptr;
96    free(job_ptr->pids);
97    free(job_ptr);
```

4.34.2.4 setup_terminal_signal_handlers()

```
void setup_terminal_signal_handlers (
                  void )
Definition at line 73 of file shell.c.
      struct sigaction sa_int = {0};
sa_int.sa_handler = shell_sigint_handler;
75
      sigemptyset(&sa_int.sa_mask);
     sa_int.sa_flags = SA_RESTART;
sigaction(SIGINT, &sa_int, NULL);
78
79
     struct sigaction sa_stp = {0};
sa_stp.sa_handler = shell_sigstp_handler;
80
82
      sigemptyset(&sa_stp.sa_mask);
83
      sa_stp.sa_flags = SA_RESTART;
     sigaction(SIGTSTP, &sa_stp, NULL);
84
85 }
```

4.34.2.5 shell()

The main shell function that runs the shell. This is run via an s_spawn call from init's process. It prompts the user for builtins and scripts to run.

Parameters

```
input unused
```

Definition at line 375 of file shell.c.

```
375
376
377
       job_list = vec_new(0, free_job_ptr);
378
379
      setup_terminal_signal_handlers();
380
381
      while (true) {
382
        int status;
383
         pid_t child_pid;
384
         while ((child_pid = s_waitpid(-1, &status, true)) > 0) {
385
         // Child process has completed, no need to do anything special
386
           // The s_waitpid function already handles cleanup
        }
387
388
389
         // prompt
         if (s_write(STDOUT_FILENO, PROMPT, strlen(PROMPT)) < 0) {</pre>
390
391
             u_perror("prompt write error");
392
393
394
         // parse user input
395
396
         char buffer[MAX_BUFFER_SIZE];
         ssize_t user_input = s_read(STDIN_FILENO, buffer, MAX_BUFFER_SIZE);
if (user_input < 0) {</pre>
397
398
399
         u_perror("shell read error");
400
           break;
        } else if (user_input == 0) { // EOF case
401
          s_shutdown_pennos();
402
403
          break;
404
405
        buffer[user_input] = '\0';
if (buffer[user_input - 1] == '\n') {
  buffer[user_input - 1] = '\0';
406
407
408
409
410
```

```
411
        struct parsed_command* cmd = NULL;
412
         int cmd_parse_res = parse_command(buffer, &cmd);
         if (cmd_parse_res != 0 || cmd == NULL) {
413
          P_ERRNO = P_EPARSE;
u_perror("parse_command");
414
415
416
          continue:
417
418
419
         \ensuremath{//} handle the command
420
         if (cmd->num_commands == 0) {
421
          free (cmd);
422
          continue;
423
424
425
        child_pid = execute_command(cmd);
        if (child_pid < 0) {</pre>
426
427
          free (cmd);
428
        continue;
} else if (child_pid == 0) {
429
430
          free(cmd);
431
          continue;
432
433
         \ensuremath{//} If background, add the process to the job list.
434
435
         if (cmd->is_background) {
          // Create a new job entry.
436
437
           job* new_job = malloc(sizeof(job));
438
           if (new_job == NULL) {
439
             perror("Error: mallocing new_job failed");
440
             free (cmd);
441
             continue:
442
443
           new_job->id = next_job_id++;
444
           new_job->pgid = child_pid; // For single commands, child's pid = pgid.
445
           new_job->num_pids = 1;
           new_job->pids = malloc(sizeof(pid_t));
if (new_job->pids == NULL) {
446
447
            perror("Error: mallocing new_job->pids failed");
448
449
             free(new_job);
450
             free (cmd);
451
             continue;
452
          new_job->pids[0] = child_pid;
new_job->state = RUNNING;
new_job->cmd = cmd; // Retain command info; do not free here.
453
454
455
456
           new_job->finished_count = 0;
457
           vec_push_back(&job_list, new_job);
458
           // Print job control information in the format: "[job_id] child_pid"
459
           char msg[128];
460
           snprintf(msg, sizeof(msg), "[%lu] %d\n", (unsigned long)new_job->id,
461
462
                     child_pid);
463
           s_write(STDOUT_FILENO, msg, strlen(msg));
        464
465
466
           current_fg_pid = child_pid;
467
           int status;
468
           s_waitpid(child_pid, &status, false);
469
           current_fg_pid = 2;
470
471
      }
472
473
      vec_destroy(&job_list);
474
      s_exit();
475
476 }
```

4.34.2.6 shell_sigint_handler()

```
57  }
58
59    s_write(STDOUT_FILENO, "\n", 1);
60 }
```

4.34.2.7 shell_sigstp_handler()

4.34.2.8 u_execute_command()

```
\label{eq:pid_tu} \mbox{pid_t u_execute\_command (} \\ \mbox{struct parsed\_command * $\it cmd$ )}
```

Helper function that will execute a given command so long as it's one of the built-ins. Notably, its output and input are determined by the spawning script process.

Parameters

cmd the parsed command to try executing

Returns

the pid of the process if one was spawned, 0 if a routine was run or -1 if not matches found

Definition at line 136 of file shell.c.

```
136
137
138
          // check for independently scheduled processes
         if (strcmp(cmd->commands[0][0], "cat") == 0) {
  return s_spawn(u_cat, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "sleep") == 0) {
139
140
141
             return s_spawn(u_sleep, cmd->commands[0], input_fd_script, output_fd_script);
142
          } else if (strcmp(cmd->commands[0][0], "busy") == 0) {
         return s_spawn(u_busy, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "echo") == 0) {
   return s_spawn(u_echo, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "ls") == 0) {
144
145
146
147
          return s_spawn(u_ls, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "touch") == 0) {
148
149
150
             return s_spawn(u_touch, cmd->commands[0], input_fd_script, output_fd_script);
         } else if (strcmp(cmd->commands[0][0], "mv") == 0) {
151
         return s_spawn(u_mv, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "cp") == 0) {
   return s_spawn(u_cp, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "rm") == 0) {
152
153
155
          return s_spawn(u_rm, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "chmod") == 0) {
156
157
```

```
return s_spawn(u_chmod, cmd->commands[0], input_fd_script, output_fd_script);
159
       } else if (strcmp(cmd->commands[0][0], "ps") == 0) {
      return s_spawn(u_ps, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "kill") == 0) {
160
161
162
      return s_spawn(u_kill, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "zombify") == 0) {
163
        return s_spawn(u_zombify, cmd->commands[0], input_fd_script, output_fd_script);
164
       } else if (strcmp(cmd->commands[0][0], "orphanify") == 0) {
165
      return s_spawn(u_orphanify, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "hang") == 0) {
   return s_spawn(hang, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "nohang") == 0) {
166
167
168
169
         return s_spawn(nohang, cmd->commands[0], input_fd_script, output_fd_script);
170
171
       } else if (strcmp(cmd->commands[0][0], "recur") == 0) {
172
         return s_spawn(recur, cmd->commands[0], input_fd_script, output_fd_script);
173
       } else if (strcmp(cmd->commands[0][0], "crash") == 0) {
         return s_spawn(crash, cmd->commands[0], input_fd_script, output_fd_script);
174
175
176
177
       // check for sub-routines
178
       if (strcmp(cmd->commands[0][0], "nice") == 0) {
179
        u_nice(cmd->commands[0]);
180
         return 0;
       } else if (strcmp(cmd->commands[0][0], "nice_pid") == 0) {
181
       u_nice_pid(cmd->commands[0]);
182
183
         return 0;
184
       } else if (strcmp(cmd->commands[0][0], "man") == 0) {
       u_man(cmd->commands[0]);
185
186
         return 0;
      } else if (strcmp(cmd->commands[0][0], "bg") == 0) {
187
188
        u bg(cmd->commands[0]);
189
         return 0;
190
       } else if (strcmp(cmd->commands[0][0], "fg") == 0) {
191
         u_fg(cmd->commands[0]);
      return 0;
} else if (strcmp(cmd->commands[0][0], "jobs") == 0) {
192
193
194
        u_jobs(cmd->commands[0]);
195
         return 0;
196
       } else if (strcmp(cmd->commands[0][0], "logout") == 0) {
197
        u_logout(cmd->commands[0]);
198
         return 0;
199
      } else {
        return -1; // no matches, no scripts now
200
201
203
       return 0;
204 }
```

4.34.2.9 u_read_and_execute_script()

Helper function that reads a script file line by line, parses each line as a command, and executes it.

Parameters

```
arg standard (function name, NULL) args
```

Returns

NULL

Definition at line 214 of file shell.c.

```
214 {
215    // read the script line by line, parse each line, and execute the command
216    while (true) {
217         char buffer[MAX_LINE_BUFFER_SIZE];
218         fill_buffer_until_full_or_newline(script_fd, buffer);
219         if (buffer[0] == '\0') {
```

```
break; // EOF case
221
222
           // parse the command
223
          // paise the command
struct parsed_command* cmd = NULL;
int parse_result = parse_command(buffer, &cmd);
if (parse_result != 0 || cmd == NULL) {
    P_ERRNO = P_EPARSE;
    u_perror("parse_command");
    free(cmd);
}
224
225
226
227
228
229
230
231
          // execute the command
232
233
          pid_t child_pid = u_execute_command(cmd);
234
          if (child_pid > 0) { // if process was spawned, wait for it to finish
235
         s_waitpid(child_pid, &status, false);
} else if (child_pid < 0) { // nothing spawning so safe to free cmd</pre>
236
237
238
239
240
241
242
        s_exit(); // exit the script
243
       return NULL;
244 }
```

4.34.3 Variable Documentation

4.34.3.1 current_fg_pid

```
pid_t current_fg_pid [extern]
```

Definition at line 31 of file kern_sys_calls.c.

4.34.3.2 input_fd_script

```
int input_fd_script = -1
```

Definition at line 42 of file shell.c.

4.34.3.3 job_list

```
Vec job_list
```

Definition at line 37 of file shell.c.

4.34.3.4 next_job_id

```
jid_t next_job_id = 1
```

Definition at line 38 of file shell.c.

4.34.3.5 output_fd_script

```
int output_fd_script = -1
```

Definition at line 43 of file shell.c.

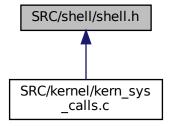
4.34.3.6 script_fd

```
int script_fd = -1
```

Definition at line 41 of file shell.c.

4.35 SRC/shell/shell.h File Reference

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



Functions

void * shell (void *input)

The main shell function that runs the shell. This is run via an s_spawn call from init's process. It prompts the user for builtins and scripts to run.

4.35.1 Function Documentation

4.35.1.1 shell()

The main shell function that runs the shell. This is run via an s_spawn call from init's process. It prompts the user for builtins and scripts to run.

Parameters

input unused

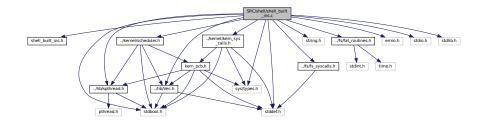
Definition at line 375 of file shell.c.

```
376
377
       job_list = vec_new(0, free_job_ptr);
378
379
      setup terminal signal handlers();
380
381
      while (true) {
382
        int status;
383
        pid_t child_pid;
384
         while ((child_pid = s_waitpid(-1, &status, true)) > 0) {
         // Child process has completed, no need to do anything special
// The s_waitpid function already handles cleanup
385
386
387
388
389
        if (s_write(STDOUT_FILENO, PROMPT, strlen(PROMPT)) < 0) {
    u_perror("prompt write error");</pre>
390
391
392
             break:
393
394
395
         // parse user input
396
         char buffer[MAX_BUFFER_SIZE];
        ssize_t user_input = s_read(STDIN_FILENO, buffer, MAX_BUFFER_SIZE);
if (user_input < 0) {</pre>
397
398
399
          u perror("shell read error");
400
          break;
        } else if (user_input == 0) { // EOF case
401
402
         s_shutdown_pennos();
403
          break;
        1
404
405
406
        buffer[user_input] = ' \setminus 0';
407
        if (buffer[user_input - 1] == '\n') {
          buffer[user_input - 1] = '\0';
408
409
410
411
        struct parsed command* cmd = NULL:
         int cmd_parse_res = parse_command(buffer, &cmd);
412
413
         if (cmd_parse_res != 0 || cmd == NULL) {
414
          P_ERRNO = P_EPARSE;
          u_perror("parse_command");
415
416
          continue:
417
418
419
         // handle the command
420
         if (cmd->num_commands == 0) {
421
         free(cmd);
422
           continue;
423
424
425
        child_pid = execute_command(cmd);
426
        if (child_pid < 0) {</pre>
427
          free (cmd);
428
          continue;
        } else if (child_pid == 0) {
429
430
          free (cmd);
431
          continue;
432
433
434
         // If background, add the process to the job list.
         if (cmd->is_background) {
435
436
          // Create a new job entry.
           job* new_job = malloc(sizeof(job));
if (new_job == NULL) {
437
438
439
             perror("Error: mallocing new_job failed");
440
             free (cmd);
441
             continue;
442
443
           new_job->id = next_job_id++;
444
           new_job->pgid = child_pid; // For single commands, child's pid = pgid.
445
           new_job->num_pids = 1;
446
           new_job->pids = malloc(sizeof(pid_t));
           if (new_job->pids == NULL) {
   perror("Error: mallocing new_job->pids failed");
447
448
449
             free (new_job);
450
             free (cmd);
451
452
```

```
453
          new_job->pids[0] = child_pid;
          new_job->state = RUNNING;
new_job->cmd = cmd; // Retain command info; do not free here.
454
455
          new_job->finished_count = 0;
456
457
          vec_push_back(&job_list, new_job);
458
          // Print job control information in the format: "[job_id] child_pid"
459
460
          char msg[128];
461
          snprintf(msg, sizeof(msg), "[\$lu] \$d\n", (unsigned long) new_job->id,
462
                    child_pid);
          s_write(STDOUT_FILENO, msg, strlen(msg));
463
464
        } else {
          // Foreground execution.
465
466
          current_fg_pid = child_pid;
467
          int status;
468
          s_waitpid(child_pid, &status, false);
469
          current_fg_pid = 2;
470
471
472
473
      vec_destroy(&job_list);
474
      s_exit();
475
      return 0;
476 }
```

4.36 SRC/shell/shell_built_ins.c File Reference

```
#include "shell_built_ins.h"
#include <stdbool.h>
#include <stddef.h>
#include <string.h>
#include <sys/types.h>
#include "../fs/fat_routines.h"
#include "../fs/fs_syscalls.h"
#include "../kernel/kern_sys_calls.h"
#include "../kernel/scheduler.h"
#include "../lib/Vec.h"
#include "../lib/spthread.h"
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
Include dependency graph for shell built ins.c:
```



Functions

```
    void * u_cat (void *arg)
        The ususal cat program.

    void * u_sleep (void *arg)
        Sleep for n seconds.
```

void * u_busy (void *arg)

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

void * u_echo (void *arg)

Echo back an input string.

void * u_ls (void *arg)

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

void * u chmod (void *arg)

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

void * u_touch (void *arg)

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

void * u_mv (void *arg)

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

- void * u cp (void *arg)
- void * u_rm (void *arg)

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

void * u_ps (void *arg)

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

void * u_kill (void *arg)

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

void * u nice (void *arg)

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

void * u_nice_pid (void *arg)

Adjust the priority level of an existing process.

void * u_man (void *arg)

Lists all available commands.

void * u_bg (void *arg)

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

void * u fg (void *arg)

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

void * u_jobs (void *arg)

Lists all jobs.

void * u_logout (void *arg)

Exits the shell and shutsdown PennOS.

void * zombie_child (void *arg)

Helper for zombify.

void * u_zombify (void *arg)

Used to test zombifying functionality of your kernel.

void * orphan_child (void *arg)

Helper for orphanify.

void * u_orphanify (void *arg)

Used to test orphanifying functionality of your kernel.

Variables

void *(*)(void *) get_associated_ufunc (char *func)

Helper function to get the associated "u-version" of a function given its standalone version. As a conrete example, if we pass in "cat", we will output "u_cat".

4.36.1 Function Documentation

4.36.1.1 orphan_child()

Helper for orphanify.

Definition at line 314 of file shell_built_ins.c.

```
314 {
315 while (1)
316 ;
317 s_exit();
318 }
```

4.36.1.2 u_bg()

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

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

Definition at line 271 of file shell_built_ins.c.

4.36.1.3 u_busy()

```
void* u_busy (
     void * arg )
```

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

Example Usage: busy

Definition at line 53 of file shell_built_ins.c.

```
53 while (1)
55 ;
56 s_exit();
57 return NULL;
58 }
```

4.36.1.4 u_cat()

```
void* u_cat (
     void * arg )
```

The ususal cat program.

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

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

Definition at line 28 of file shell_built_ins.c.

4.36.1.5 u_chmod()

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

Print appropriate error message if:

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

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

Definition at line 72 of file shell_built_ins.c.

4.36.1.6 u_cp()

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

Print appropriate error message if:

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

Example Usage: cp src_file dst_file

Definition at line 90 of file shell_built_ins.c.

```
90 {
91 cp(arg);
92 s_exit();
93 return NULL;
94 }
```

4.36.1.7 u echo()

```
void* u_echo (
          void * arg )
```

Echo back an input string.

Example Usage: echo Hello World

Definition at line 60 of file shell_built_ins.c.

4.36.1.8 u_fg()

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

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

Definition at line 276 of file shell_built_ins.c.

```
276 {
277  // TODO --> implement fg
278  return NULL;
279 }
```

4.36.1.9 u_jobs()

Lists all jobs.

Example Usage: jobs

Definition at line 281 of file shell built ins.c.

```
281 {
282    // TODO --> implement jobs
283    return NULL;
284 }
```

4.36.1.10 u kill()

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

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

Definition at line 108 of file shell_built_ins.c.

```
108
109
      char** argv = (char**)arg;
     110
111
     char err_buf[128];
112
113
114
     // Check if the first argument specifies a signal
115
     if (argv[start_index] && argv[start_index][0] == '-') {
116
          (strcmp(argv[start_index], "-term") == 0) {
117
         sig = 2;
       } else if (strcmp(argv[start_index], "-stop") == 0) {
118
         sig = 0;
119
       } else if (strcmp(argv[start_index], "-cont") == 0) {
120
121
         sig = 1;
122
       } else {
123
         // Construct error message
124
         s_exit();
         return NULL:
125
126
127
       start_index++;
128
129
130
     // Process each PID argument using strtol
     for (int i = start_index; argv[i] != NULL; i++) {
131
132
       char* endptr;
133
       long pid_long = strtol(argv[i], &endptr, 10);
       if (*endptr!= '\0' || pid_long <= 0) {
    snprintf(err_buf, 128, "Invalid PID: %s\n", argv[i]);</pre>
134
135
136
         s_write(STDERR_FILENO, err_buf, strlen(err_buf));
137
         continue;
138
139
       pid_t pid = (pid_t)pid_long;
       if (s_kill(pid, sig) < 0)</pre>
140
         snprintf(err_buf, 128, "b_kill error on PID d\n, pid);
141
142
         s_write(STDERR_FILENO, err_buf, strlen(err_buf));
143
144
     }
145
     s exit();
146
     return NULL;
147 }
```

4.36.1.11 u_logout()

Exits the shell and shutsdown PennOS.

Example Usage: logout

Definition at line 286 of file shell_built_ins.c.

4.36.1.12 u_ls()

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

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

Definition at line 66 of file shell_built_ins.c.

```
66
67 ls(arg);
68 s_exit();
69 return NULL;
70 }
```

4.36.1.13 u_man()

Lists all available commands.

Example Usage: man

Definition at line 233 of file shell_built_ins.c.

```
234
235
236
237
238
239
             "ls : lists all files in the working directory\n"
"touch f1 f2 ... : for each file, creates empty file if it doesn't "
240
241
             "exist yet, otherwise updates its timestamp\n"
"mv f1 f2 : renames f1 to f2 (overwrites f2 if it exists)\n"
"cp f1 f2 : copies f1 to f2 (overwrites f2 if it exists)\n"
242
243
244
            "rm f1 f2 ... : removes the input list of files\n"
"chmod +_ f1 : changes f1 permissions to +_ specifications "
245
246
             "(+x, +rw, etc)\n"
247
             "ps
                                         : lists all processes on PennOS, displaying PID, "
2.48
            "PPID, priority, status, and command name\n"
"kill (-__) pidl pid 2 : sends specified signal (term default) to list "
249
250
             "of processes\n"
```

```
252
          "nice n command
                                 : spawns a new process for command and sets its "
253
          "priority to n \ "
254
          "nice_pid n pid
                                 : adjusts the priority level of an existing "
          "process to n \ n"
2.5.5
                                 : lists all available commands in PennOS\n"
256
          "man
257
          "bq
                                  : resumes most recently stopped process in "
          "background or the one specified by job_id\n"
259
          "fg
                                  : brings most recently stopped or background job "
260
          "to foreground or the one specifed by job_id\n"
                    : lists all jobs\n"
: exits the shell and shuts down PennOS\n"
261
          "jobs
          "logout
262
          "zombify
                                 : creates a child process that becomes a zombie\n"
263
264
          "orphanify
                                 : creates a child process that becomes an '
265
          "orphan\n";
266
267
     s_write(STDOUT_FILENO, man_string, strlen(man_string));
268
     return NULL:
269 }
```

4.36.1.14 u mv()

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

Print appropriate error message if:

- src_file is not a file that exists
- src file does not have read permissions
- dst file file already exists but does not have write permissions

Example Usage: mv src_file dst_file

Definition at line 84 of file shell built ins.c.

```
84 {
85 mv(arg);
86 s_exit();
87 return NULL;
88 }
```

4.36.1.15 u_nice()

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

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

Definition at line 194 of file shell_built_ins.c.

```
202
203
      char* command = ((char**)arg)[2];
204
      void* (*ufunc)(void*) = get_associated_ufunc(command);
      if (ufunc == NULL) {
205
206
        return NULL; // no matches, don't spawn
207
208
209
      pid_t new_proc_pid = s_spawn(ufunc, &((char**)arg)[2], 0, 1);
210
      if (new_proc_pid != -1) { // non-error case
211
     . ..._ploc_pia := -1) { // non-err
s_nice(new_proc_pid, new_priority);
}
212
213
214
215 return NULL;
216 }
```

4.36.1.16 u_nice_pid()

Adjust the priority level of an existing process.

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

Definition at line 218 of file shell_built_ins.c.

```
219
       char* endptr;
220
       errno = 0;
       int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10); if (*endptr != '\0' || errno != 0) { // error catch
221
222
         return NULL;
223
224
      pid_t pid = (pid_t)strtol(((char**)arg)[2], &endptr, 10);
if (*endptr != '\0' || errno != 0) {
225
226
      return NULL;
}
227
228
229
       s_nice(pid, new_priority);
230
       return NULL;
231 }
```

4.36.1.17 u_orphanify()

Used to test orphanifying functionality of your kernel.

Example Usage: orphanify

Definition at line 320 of file shell_built_ins.c.

```
321 char* orphan_child_argv[] = {"orphan_child", NULL};
322 s_spawn(orphan_child, orphan_child_argv, STDIN_FILENO, STDOUT_FILENO);
323 s_exit();
324 return NULL;
325 }
```

4.36.1.18 u_ps()

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

Example Usage: ps

Definition at line 102 of file shell_built_ins.c.

4.36.1.19 u_rm()

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

Print appropriate error message if:

• file is not a file that exists

Example Usage: rm f1 f2 f3 f4 f5

Definition at line 96 of file shell_built_ins.c.

```
96 {
97 rm(arg);
98 s_exit();
99 return NULL;
100 }
```

4.36.1.20 u_sleep()

Sleep for n seconds.

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

Example Usage: sleep 10

Definition at line 34 of file shell_built_ins.c.

```
34
35
    char* endptr;
    errno = 0;
37
    if (((char**)arg)[1] == NULL) { // no arg case
38
      s_exit();
      return NULL;
39
40
    int sleep_secs = (int)strtol(((char**)arg)[1], &endptr, 10);
    if (*endptr != '\0' || errno != 0 || sleep_secs <= 0) {</pre>
     s_exit();
44
      return NULL;
45
    }
46
    int sleep_ticks = sleep_secs * 10;
   s_sleep(sleep_ticks);
49
    s_exit();
    return NULL;
50
51 }
```

4.36.1.21 u_touch()

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

Example Usage: touch f1 f2 f3 f4 f5

Definition at line 78 of file shell_built_ins.c.

4.36.1.22 u_zombify()

```
void* u_zombify (
     void * arg )
```

Used to test zombifying functionality of your kernel.

Example Usage: zombify

Definition at line 303 of file shell_built_ins.c.

4.36.1.23 zombie_child()

Helper for zombify.

Definition at line 298 of file shell built ins.c.

4.36.2 Variable Documentation

4.36.2.1 get_associated_ufunc

Helper function to get the associated "u-version" of a function given its standalone version. As a conrete example, if we pass in "cat", we will output "u_cat".

Parameters

func A string of the function name to get the associated ufunc for

Returns

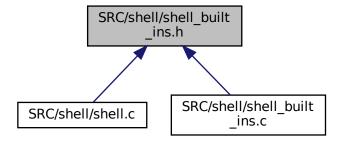
192 }

A ptr to the associated u-version function or NULL if no matches are found

```
Definition at line 164 of file shell_built_ins.c.
      if (strcmp(func, "cat") == 0) {
165
     return u_cat;
} else if (strcmp(func, "sleep") == 0) {
166
167
168
       return u sleep;
     } else if (strcmp(func, "busy") == 0) {
169
        return u_busy;
171
     } else if (strcmp(func, "echo") == 0) {
     return u_echo;
} else if (strcmp(func, "ls") == 0) {
172
173
174
        return u_ls;
     } else if (strcmp(func, "touch") == 0) {
176
       return u_touch;
177
     } else if (strcmp(func, "mv") == 0) {
     return u_mv;
} else if (strcmp(func, "cp") == 0) {
178
179
180
        return u cp;
     } else if (strcmp(func, "rm") == 0) {
181
        return u_rm;
     } else if (strcmp(func, "chmod") == 0) {
183
184
        return u_chmod;
185
     } else if (strcmp(func, "ps") == 0) {
186
        return u_ps;
187
     } else if (strcmp(func, "kill") == 0) {
188
       return u_kill;
189
190
     return NULL; // no matches case
191
```

4.37 SRC/shell/shell built ins.h File Reference

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



Functions

```
void * u_cat (void *arg)
      The ususal cat program.
void * u_sleep (void *arg)
      Sleep for n seconds.
void * u_busy (void *arg)
      Busy wait indefinitely. It can only be interrupted via signals.
void * u echo (void *arg)
      Echo back an input string.
void * u_ls (void *arg)
      Lists all files in the working directory. For extra credit, it should support relative and absolute file paths.
void * u_touch (void *arg)
      For each file, create an empty file if it doesn't exist, else update its timestamp.
void * u mv (void *arg)
      Rename a file. If the dst_file file already exists, overwrite it.
void * u cp (void *arg)
void * u_rm (void *arg)
      Remove a list of files. Treat each file in the list as a separate transaction. (i.e. if removing file1 fails, still attempt to
      remove file2, file3, etc.)

    void * u chmod (void *arg)

      Change permissions of a file. There's no need to error if a permission being added already exists, or if a permission
      being removed is already not granted.
void * u_ps (void *arg)
      List all processes on PennOS, displaying PID, PPID, priority, status, and command name.
void * u kill (void *arg)
      Sends a specified signal to a list of processes. If a signal name is not specified, default to "term". Valid signals are
      -term, -stop, and -cont.

    void * u nice (void *arg)

      Spawn a new process for command and set its priority to priority.
void * u_nice_pid (void *arg)
      Adjust the priority level of an existing process.

    void * u man (void *arg)

      Lists all available commands.
void * u_bg (void *arg)
      Resumes the most recently stopped job in the background, or the job specified by job_id.

    void * u fg (void *arg)

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

    void * u logout (void *arg)

      Exits the shell and shutsdown PennOS.

    void * u_zombify (void *arg)

      Used to test zombifying functionality of your kernel.
```

4.37.1 Function Documentation

void * u_orphanify (void *arg)

Used to test orphanifying functionality of your kernel.

4.37.1.1 u_bg()

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

Example Usage: bg Example Usage: bg 2 (job id is 2)

Definition at line 271 of file shell built ins.c.

4.37.1.2 u_busy()

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

Example Usage: busy

Definition at line 53 of file shell_built_ins.c.

```
53 {
54 while (1)
55 ;
56 s_exit();
57 return NULL;
```

4.37.1.3 u_cat()

The ususal cat program.

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

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

Definition at line 28 of file shell_built_ins.c.

```
28
29 cat(arg);
30 s_exit();
31 return NULL;
```

4.37.1.4 u_chmod()

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

Print appropriate error message if:

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

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

Definition at line 72 of file shell_built_ins.c.

4.37.1.5 u_cp()

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

Print appropriate error message if:

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

Example Usage: cp src file dst file

Definition at line 90 of file shell_built_ins.c.

```
90 {
91 cp(arg);
92 s_exit();
93 return NULL;
94 }
```

4.37.1.6 u_echo()

```
void* u_echo (
          void * arg )
```

Echo back an input string.

Example Usage: echo Hello World

Definition at line 60 of file shell_built_ins.c.

```
60 {
61    s_echo(arg);
62    s_exit();
63    return NULL;
64 }
```

4.37.1.7 u_fg()

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

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

Definition at line 276 of file shell_built_ins.c.

```
276 {
277  // TODO --> implement fg
278  return NULL;
279 }
```

4.37.1.8 u_jobs()

```
void* u_jobs (
     void * arg )
```

Lists all jobs.

Example Usage: jobs

Definition at line 281 of file shell_built_ins.c.

```
281 {
282  // TODO --> implement jobs
283  return NULL;
284 }
```

4.37.1.9 u_kill()

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

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

Definition at line 108 of file shell_built_ins.c.

```
109
       char** argv = (char**)arg;
       110
111
       char err_buf[128];
112
113
114
       \ensuremath{//} Check if the first argument specifies a signal
115
       if (argv[start_index] && argv[start_index][0] == '-') {
   if (strcmp(argv[start_index], "-term") == 0) {
116
117
           sig = 2;
         } else if (strcmp(argv[start_index], "-stop") == 0) {
118
119
           sig = 0;
120
         } else if (strcmp(argv[start_index], "-cont") == 0) {
121
            sig = 1;
         } else {
  // Construct error message
122
123
124
           s exit():
125
            return NULL;
126
127
         start_index++;
128
129
130
       // Process each PID argument using strtol
131
       for (int i = start_index; argv[i] != NULL; i++) {
132
        char* endptr;
        long pid_long = strtol(argv[i], &endptr, 10);
if (*endptr != '\0' || pid_long <= 0) {
    snprintf(err_buf, 128, "Invalid PID: %s\n", argv[i]);</pre>
133
134
135
136
           s_write(STDERR_FILENO, err_buf, strlen(err_buf));
137
           continue;
138
139
        pid_t pid = (pid_t)pid_long;
         if (s_kill(pid, sig) < 0) {
    snprintf(err_buf, 128, "b_kill error on PID %d\n", pid);
    s_write(STDERR_FILENO, err_buf, strlen(err_buf));</pre>
140
141
142
143
144
      }
145
       s_exit();
146
      return NULL;
147 }
```

4.37.1.10 u logout()

Exits the shell and shutsdown PennOS.

Example Usage: logout

Definition at line 286 of file shell_built_ins.c.

```
286 {
287     s_shutdown_pennos();
288     return NULL;
289 }
```

4.37.1.11 u_ls()

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

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

Definition at line 66 of file shell built ins.c.

```
66 {
67    ls(arg);
68    s_exit();
69    return NULL;
70 }
```

4.37.1.12 u_man()

Lists all available commands.

Example Usage: man

Definition at line 233 of file shell_built_ins.c.

```
233
      const char* man_string =
234
235
           "cat f1 f2 ...
                                   : concatenates provided files (if none, reads from "
236
           "std in), and writes to std out\n"
                          : sleeps for n seconds\n"
237
           "sleep n
238
           "busy
                                    : busy waits indefinitely\n"
           "echo str
239
                                    : echoes back the input string \mathsf{str} \setminus \mathsf{n}"
           "Is : lists all files in the working directory\n"
"touch f1 f2 ... : for each file, creates empty file if it doesn't "
240
241
           "exist yet, otherwise updates its timestamp\n"
242
                          : renames f1 to f2 (overwrites f2 if it exists)\n"
: copies f1 to f2 (overwrites f2 if it exists)\n"
243
           "mv f1 f2
244
           "cp f1 f2
           "rm f1 f2 ...
"chmod +_ f1
245
                                    : removes the input list of files\n"
246
                                    : changes fl permissions to +_ specifications "
247
           "(+x, +rw, etc)\n"
248
           "ps
                                    : lists all processes on PennOS, displaying PID, "
249
           "PPID, priority, status, and command name\n"
250
           "kill (-__) pid1 pid 2 : sends specified signal (term default) to list "
251
           "of processes\n"
           "nice n command
252
                                    : spawns a new process for command and sets its "
253
           "priority to n\n'
254
           "nice_pid n pid
                                    : adjusts the priority level of an existing
255
           "process to n \ n"
256
                                    : lists all available commands in PennOS\n"
           "man
                                    : resumes most recently stopped process in "
257
           "bg
           "background or the one specified by job_id\n"
"fg : brings most recently stopped or background job"
258
259
           "to foreground or the one specifed by job_id\n"
260
261
           "jobs
                                   : lists all jobs\n"
262
           "logout
                                    : exits the shell and shuts down PennOS\n"
263
           "zombify
                                    : creates a child process that becomes a zombie \n"
           "orphanify
264
                                   : creates a child process that becomes an
265
           "orphan\n";
266
      s_write(STDOUT_FILENO, man_string, strlen(man_string));
267
268
269 }
```

4.37.1.13 u_mv()

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

Print appropriate error message if:

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

Example Usage: mv src_file dst_file

Definition at line 84 of file shell_built_ins.c.

```
84 {
85 mv(arg);
86 s_exit();
87 return NULL;
88 }
```

4.37.1.14 u_nice()

```
void* u_nice (
     void * arg )
```

Spawn a new process for ${\tt command}$ and set its priority to ${\tt priority}.$

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

Definition at line 194 of file shell built ins.c.

```
194
195
       char* endptr;
196
       errno = 0;
       int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10);
if (*endptr != '\0' || errno != 0 || new_priority > 2 ||
    new_priority < 0) { // error catch
    return NULL:</pre>
197
198
199
200
        return NULL;
201
202
203
       char* command = ((char**)arg)[2];
       void* (*ufunc)(void*) = get_associated_ufunc(command);
204
205
       if (ufunc == NULL) {
         return NULL; // no matches, don't spawn
206
207
208
209
       pid_t new_proc_pid = s_spawn(ufunc, &((char**)arg)[2], 0, 1);
210
       if (new_proc_pid != -1) { // non-error case
211
212
        s_nice(new_proc_pid, new_priority);
213
214
215
216 }
       return NULL;
```

4.37.1.15 u_nice_pid()

Adjust the priority level of an existing process.

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

Definition at line 218 of file shell_built_ins.c.

```
218
219
         char* endptr;
220
        errno = 0;
        int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10); if (*endptr != '\0' || errno != 0) { // error catch
221
222
          return NULL;
223
224
        pid_t pid = (pid_t)strtol(((char**)arg)[2], &endptr, 10);
if (*endptr != '\0' || errno != 0) {
    return NULL;
225
226
227
228
229 s_nice(pid, new_priority);
230 return NULL;
231 }
```

4.37.1.16 u_orphanify()

Used to test orphanifying functionality of your kernel.

Example Usage: orphanify

Definition at line 320 of file shell_built_ins.c.

4.37.1.17 u_ps()

```
void* u_ps (
     void * arg )
```

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

Example Usage: ps

Definition at line 102 of file shell_built_ins.c.

{

4.37.1.18 u_rm()

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

Print appropriate error message if:

• file is not a file that exists

Example Usage: rm f1 f2 f3 f4 f5

Definition at line 96 of file shell_built_ins.c.

```
96 {
97 rm(arg);
98 s_exit();
99 return NULL;
100 }
```

4.37.1.19 u sleep()

```
void* u_sleep (
     void * arg )
```

Sleep for n seconds.

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

Example Usage: sleep 10

Definition at line 34 of file shell built ins.c.

```
35
      char* endptr;
     errno = 0;
if (((char**)arg)[1] == NULL) { // no arg case
36
37
38
       s_exit();
39
        return NULL;
     int sleep_secs = (int)strtol(((char**)arg)[1], &endptr, 10);
if (*endptr != '\0' || errno != 0 || sleep_secs <= 0) {
41
43
        s_exit();
        return NULL;
44
45
     int sleep_ticks = sleep_secs * 10;
     s_sleep(sleep_ticks);
48
49
     s_exit();
    return NULL;
50
51 }
```

4.37.1.20 u_touch()

```
void* u_touch (
            void * arg )
```

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

Example Usage: touch f1 f2 f3 f4 f5

Definition at line 78 of file shell_built_ins.c.

```
couch(arg);
80    s_exit();
81    return NULL;
82 }
 79
          touch (arg);
```

4.37.1.21 u_zombify()

```
void* u_zombify (
          void * arg )
```

Used to test zombifying functionality of your kernel.

Example Usage: zombify

Definition at line 303 of file shell_built_ins.c.

```
303 {
304 char* zombie_child_argv[] = {"zombie_child", NULL};
305 s_spawn(zombie_child, zombie_child_argv, STDIN_FILENO, STDOUT_FILENO);
306 while (1)
307
307 ;
308 return NULL;
309 }
```

Index

_GNU_SOURCE	fat_routines.c, 28
spthread.c, 195	fat_routines.h, 45
	commands
ack	parsed_command, 11
spthread_signal_args_st, 18	compact_directory
actual_arg	fs_helpers.c, 59
spthread_fwd_args_st, 16	fs_helpers.h, 75
actual_routine	CONT_BY_SIG
spthread_fwd_args_st, 16	signal.h, 183
add_file_entry	copy_host_to_pennfat
fs_helpers.c, 57	fs_helpers.c, 61
fs_helpers.h, 72	fs_helpers.h, 77
alarm_handler	copy_pennfat_to_host
scheduler.c, 163	fs_helpers.c, 63
scheduler.h, 173	fs_helpers.h, 79
allocate_block	copy_source_to_dest
fs_helpers.c, 58	fs_helpers.c, 64
fs_helpers.h, 74	fs helpers.h, 80
	ср
block_size	fat routines.c, 28
fs_helpers.c, 69	fat routines.h, 46
fs_helpers.h, 87	crash
builtins.c	stress.c, 186
u_perror, 231	stress.h, 187
builtins.h	create_pcb
u_perror, 233	kern_pcb.c, 124
	kern_pcb.h, 129
capacity	curr_priority_arr_index
vec_st, 20	scheduler.c, 170
cat	current_fg_pid
fat_routines.c, 24	fs kfuncs.c, 99
fat_routines.h, 41	kern_sys_calls.c, 142
child_in_zombie_queue	shell.c, 256
scheduler.c, 163	current_pcbs
scheduler.h, 174	kern_pcb.c, 127
child_meta	kern_sys_calls.c, 142
spthread_fwd_args_st, 16	scheduler.c, 170
child_pcbs	current_running_pcb
pcb_st, 13	fs kfuncs.c, 99
child_with_changed_process_status	kern pcb.c, 127
scheduler.c, 164	kern_sys_calls.c, 142
scheduler.h, 174	scheduler.c, 170
chmod	Scrieduler.c, 170
fat_routines.c, 26	data
fat_routines.h, 43	vec_st, 21
cmd	decrement fd ref count
job_st, 9	fs_helpers.c, 65
cmd_str	fs_helpers.h, 81
pcb_st, 13	delete_from_explicit_queue
cmpctdir	kern sys calls c 135

kern_sys_calls.h, 145	mkfs, 30
delete_from_queue	mount, 31
kern_sys_calls.c, 135	mv, 32
kern_sys_calls.h, 146	rm, 34
delete_process_from_all_queues	touch, 35
scheduler.c, 164	unmount, 36
scheduler.h, 175	fat routines.h
delete process from all queues except current	_ cat, 41
scheduler.c, 164	chmod, 43
scheduler.h, 175	cmpctdir, 45
delete_process_from_particular_queue	cp, 46
scheduler.c, 165	F_APPEND, 38
scheduler.h, 176	F READ, 38
det_priorities_arr	F WRITE, 38
scheduler.c, 171	FAT EOF, 39
determine_index_in_queue	FAT FREE, 39
kern sys calls.c, 135	ls, 47
— • — ·	
kern_sys_calls.h, 146	mkfs, 48
dir_entry_t, 5	mount, 50
firstBlock, 5	mv, 51
mtime, 5	PERM_EXEC, 39
name, 6	PERM_NONE, 39
perm, 6	PERM_READ, 39
reserved, 6	PERM_READ_EXEC, 39
size, 6	PERM_READ_WRITE, 40
type, 6	PERM_READ_WRITE_EXEC, 40
	PERM_WRITE, 40
ele_dtor_fn	rm, 52
vec_st, 21	touch, 53
execute_command	TYPE_DIRECTORY, 40
shell.c, 249	TYPE_REGULAR, 40
EXITED_NORMALLY	TYPE_SYMLINK, 40
signal.h, 183	TYPE_UNKNOWN, 41
EXPECT_COMMANDS	unmount, 54
parser.h, 242	fat size
EXPECT_INPUT_FILENAME	fs_helpers.c, 69
parser.h, 242	fs_helpers.h, 87
EXPECT_OUTPUT_FILENAME	fd_entry_t, 7
parser.h, 243	filename, 7
	first_block, 7
F_APPEND	in_use, 7
fat_routines.h, 38	mode, 7
F_READ	position, 8
fat_routines.h, 38	ref_count, 8
F_WRITE	size, 8
fat routines.h, 38	fd_table
fat	
fs_helpers.c, 69	fs_helpers.c, 70
fs_helpers.h, 87	fs_helpers.h, 87
FAT EOF	pcb_st, 13
fat_routines.h, 39	FILE_DESCRIPTOR_TABLE_SIZE
FAT FREE	kern_pcb.h, 129
fat_routines.h, 39	filename
fat_routines.c	fd_entry_t, 7
cat, 24	fill_buffer_until_full_or_newline
chmod, 26	shell.c, 251
cmpctdir, 28	find_file
•	fs_helpers.c, 65
cp, 28	fs_helpers.h, 82
ls, 29	

84

FINISHED	has_executable_permission,
Job.h, 236	increment fd ref count, 85
finished_count	init_fd_table, 85
job_st, 9	is_mounted, 88
first block	mark_entry_as_deleted, 86
_	MAX FDS, 88
fd_entry_t, 7	-
firstBlock	num_fat_blocks, 88
dir_entry_t, 5	fs_kfuncs.c
format_file_info	current_fg_pid, 99
fs_kfuncs.c, 89	current_running_pcb, 99
free_job_ptr	format_file_info, 89
shell.c, 251	k_close, 89
free_pcb	k_ls, 90
kern_pcb.c, 124	k_lseek, 91
kern_pcb.h, 130	k_open, 92
free_scheduler_queues	k_read, 94
scheduler.c, 165	k_unlink, 95
scheduler.h, 176	k_write, 96
fs_fd	fs_kfuncs.h
fs_helpers.c, 70	k_close, 101
fs_helpers.h, 87	k_ls, 102
fs_helpers.c	k_lseek, 103
add_file_entry, 57	k_open, 105
allocate_block, 58	k_read, 107
block_size, 69	k_unlink, 109
compact_directory, 59	k_write, 110
copy_host_to_pennfat, 61	SEEK CUR, 101
copy_pennfat_to_host, 63	SEEK END, 101
	SEEK SET, 101
copy_source_to_dest, 64	- ·
decrement_fd_ref_count, 65	fs_syscalls.c
fat, 69	s_close, 114
fat_size, 69	s_ls, 115
fd_table, 70	s_lseek, 115
find_file, 65	s_open, 115
fs_fd, 70	s_read, 116
get_free_fd, 66	s_unlink, 116
has_executable_permission, 67	s_write, 116
increment_fd_ref_count, 67	fs_syscalls.h
init_fd_table, 68	s_close, 118
is mounted, 70	s_ls, 119
mark entry as deleted, 68	s Iseek, 119
	- ·
MAX_FDS, 70	s_open, 120
num_fat_blocks, 70	s_read, 121
fs_helpers.h	s_unlink, 122
add_file_entry, 72	s_write, 122
allocate_block, 74	STDERR_FILENO, 118
block_size, 87	STDIN_FILENO, 118
compact_directory, 75	STDOUT_FILENO, 118
copy_host_to_pennfat, 77	
copy_pennfat_to_host, 79	generate_next_priority
copy_source_to_dest, 80	scheduler.c, 165
decrement_fd_ref_count, 81	scheduler.h, 176
fat, 87	get_associated_ufunc
•	shell_built_ins.c, 269
fat_size, 87	get_free_fd
fd_table, 87	fs_helpers.c, 66
find_file, 82	fs_helpers.h, 83
fs_fd, 87	
get_free_fd, 83	get_next_pcb
	scheduler.c, 166

scheduler.h, 177	cmd, 9
get_pcb_in_queue	finished_count, 9
scheduler.c, 166	id, 9
scheduler.h, 178	num_pids, 9
	pgid, 9
handle_signal	pids, 10
scheduler.c, 167	state, 10
scheduler.h, 178	job_state_t
hang	Job.h, 236
stress.c, 186	JUMP_OUT
stress.h, 187	parser.c, 237
has_executable_permission	parsons, 207
fs_helpers.c, 67	k close
fs_helpers.h, 84	fs_kfuncs.c, 89
10_1101p010.11, 0 1	fs_kfuncs.h, 101
id	k ls
job_st, 9	fs kfuncs.c, 90
in_use	fs kfuncs.h, 102
	-
fd_entry_t, 7	k_lseek
increment_fd_ref_count	fs_kfuncs.c, 91
fs_helpers.c, 67	fs_kfuncs.h, 103
fs_helpers.h, 85	k_open
init_fd_table	fs_kfuncs.c, 92
fs_helpers.c, 68	fs_kfuncs.h, 105
fs_helpers.h, 85	k_proc_cleanup
init_func	kern_pcb.c, 125
kern_sys_calls.c, 136	kern_pcb.h, 131
kern_sys_calls.h, 147	k_proc_create
initialize_scheduler_queues	kern_pcb.c, 126
scheduler.c, 167	kern_pcb.h, 132
scheduler.h, 179	k read
input fd	fs kfuncs.c, 94
pcb_st, 13	fs_kfuncs.h, 107
• —	
input_fd_script shell.c, 256	k_unlink
	fs_kfuncs.c, 95
is_background	fs_kfuncs.h, 109
parsed_command, 11	k_write
is_file_append	fs_kfuncs.c, 96
parsed_command, 11	fs_kfuncs.h, 110
is_mounted	kern_pcb.c
fs_helpers.c, 70	create_pcb, 124
fs_helpers.h, 88	current_pcbs, 127
is_sleeping	current_running_pcb, 127
pcb_st, 13	free_pcb, 124
	k_proc_cleanup, 125
jid_t	k_proc_create, 126
Job.h, 235	next_pid, 127
job	remove_child_in_parent, 126
Job.h, 236	kern pcb.h
Job.h	create pcb, 129
FINISHED, 236	—
jid_t, 235	FILE_DESCRIPTOR_TABLE_SIZE, 129
job, 236	free_pcb, 130
job_state_t, 236	k_proc_cleanup, 131
• — —	k_proc_create, 132
RUNNING, 236	pcb_t, 129
STOPPED, 236	remove_child_in_parent, 133
job_list	kern_sys_calls.c
shell.c, 256	current_fg_pid, 142
job_st, 8	current_pcbs, 142

current_running_pcb, 142	log_fd, 161
delete_from_explicit_queue, 135	log_generic_event, 159
delete_from_queue, 135	log_nice_event, 160
determine_index_in_queue, 135	log_scheduling_event, 161
init_func, 136	tick_counter, 161
move_pcb_correct_queue, 136	Is
one_priority_queue, 143	fat_routines.c, 29
s_cleanup_init_process, 137	fat_routines.h, 47
s_echo, 137	
s_exit, 138	main
s_kill, 138	pennfat.c, 227
s_nice, 138	pennos.c, 230
s_ps, 139	mark_entry_as_deleted
s_sleep, 139	fs_helpers.c, 68
s_spawn, 140	fs_helpers.h, 86
s_spawn_init, 140	MAX_BUFFER_SIZE
s_waitpid, 141	shell.c, 248
sleep_blocked_queue, 143	MAX_FDS
tick_counter, 143	fs_helpers.c, 70
two_priority_queue, 143	fs_helpers.h, 88
zero_priority_queue, 143	MAX_LINE_BUFFER_SIZE
zombie_queue, 143	shell.c, 249
kern_sys_calls.h	meta
delete_from_explicit_queue, 145	spthread_st, 20
delete_from_queue, 146	meta_mutex
determine_index_in_queue, 146	spthread_meta_st, 17
init_func, 147	MILISEC_IN_NANO
move_pcb_correct_queue, 147	spthread.c, 195
s_cleanup_init_process, 148	mkfs
s_echo, 148	fat_routines.c, 30
s_exit, 149	fat_routines.h, 48
s_kill, 149	mode
s_nice, 151	fd_entry_t, 7
s_ps, 151	mount
s_sleep, 152	fat_routines.c, 31
s_spawn, 153	fat_routines.h, 50
s_spawn_init, 153	move_pcb_correct_queue
s_waitpid, 154	kern_sys_calls.c, 136
length	kern_sys_calls.h, 147 mtime
vec st, 21	
log_fd	dir_entry_t, 5 mv
logger.h, 161	fat routines.c, 32
pennos.c, 230	fat routines.h, 51
scheduler.c, 171	lat_routilles.ii, 51
log_generic_event	name
logger.c, 156	dir_entry_t, 6
logger.h, 159	next_job_id
log_nice_event	shell.c, 256
logger.c, 157	next_pid
logger.h, 160	kern_pcb.c, 127
log_scheduling_event	nohang
logger.c, 158	stress.c, 186
logger.h, 161	stress.h, 188
logger.c	num_commands
log_generic_event, 156	parsed_command, 11
log_nice_event, 157	num_fat_blocks
log_scheduling_event, 158	fs_helpers.c, 70
logger.h	fs_helpers.h, 88
55	, -

num_pids	pennos-errno.h, 193
job_st, 9	P INITFAIL
· — ·	pennos-errno.h, 193
one_priority_queue	P NEEDF
kern_sys_calls.c, 143	_
scheduler.c, 171	pennos-errno.h, 193
•	P_SIGCONT
orphan_child	signal.h, 183
shell_built_ins.c, 261	P_SIGSTOP
output_fd	signal.h, 184
pcb_st, 14	P SIGTERM
output_fd_script	signal.h, 184
shell.c, 256	P WIFEXITED
5	-
P EBADF	signal.h, 184
pennos-errno.h, 190	P_WIFSIGNALED
P EBUSY	signal.h, 184
	P_WIFSTOPPED
pennos-errno.h, 190	signal.h, 184
P_ECLOSE	par pid
pennos-errno.h, 190	pcb_st, 14
P_ECOMMAND	parse_command
pennos-errno.h, 190	• —
P EEXIST	parser.c, 237
pennos-errno.h, 190	parser.h, 244
P_EFS_NOT_MOUNTED	parsed_command, 10
	commands, 11
pennos-errno.h, 190	is_background, 11
P_EFULL	is_file_append, 11
pennos-errno.h, 191	num_commands, 11
P_EFUNC	stdin_file, 11
pennos-errno.h, 191	stdout_file, 11
P_EINTR	parser.c
	parser.c
pennos-errno.h, 191	
pennos-errno.h, 191 P EINVAL	JUMP_OUT, 237
P_EINVAL	JUMP_OUT, 237 parse_command, 237
P_EINVAL pennos-errno.h, 191	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240
P_EINVAL pennos-errno.h, 191 P_ELSEEK	JUMP_OUT, 237 parse_command, 237
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_PIPELINE, 243
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EOPEN	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EPARSE pennos-errno.h, 192	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_PIPELINE, 243
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EOPEN	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 pcb_st, 12 child_pcbs, 13
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EPARSE pennos-errno.h, 192 P_EPERM pennos-errno.h, 192	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_PIPELINE, 243 pcb_st, 12 child_pcbs, 13 cmd_str, 13
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EPARSE pennos-errno.h, 192 P_EPERM	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_PIPELINE, 243 pcb_st, 12 child_pcbs, 13 cmd_str, 13 fd_table, 13
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EPARSE pennos-errno.h, 192 P_EPERM pennos-errno.h, 192	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_PIPELINE, 243 pcb_st, 12 child_pcbs, 13 cmd_str, 13 fd_table, 13 input_fd, 13
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EPARSE pennos-errno.h, 192 P_EPARSE pennos-errno.h, 192 P_EPERM pennos-errno.h, 192 P_EPERM pennos-errno.h, 192 P_EREAD	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_PIPELINE, 243 pcb_st, 12 child_pcbs, 13 cmd_str, 13 fd_table, 13 input_fd, 13 is_sleeping, 13
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EPARSE pennos-errno.h, 192 P_EPERM pennos-errno.h, 192 P_EPERM pennos-errno.h, 192 P_EREAD pennos-errno.h, 193 P_ERRNO	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_PIPELINE, 243 pcb_st, 12 child_pcbs, 13 cmd_str, 13 fd_table, 13 input_fd, 13 is_sleeping, 13 output_fd, 14
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EPARSE pennos-errno.h, 192 P_EPERM pennos-errno.h, 192 P_EPERM pennos-errno.h, 193 P_EREAD pennos-errno.h, 193 P_ERRNO pennos-errno.c, 189	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_PIPELINE, 243 pcb_st, 12 child_pcbs, 13 cmd_str, 13 fd_table, 13 input_fd, 13 is_sleeping, 13 output_fd, 14 par_pid, 14
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EPARSE pennos-errno.h, 192 P_EPERM pennos-errno.h, 192 P_EPERM pennos-errno.h, 193 P_EREAD pennos-errno.h, 193 P_ERRNO pennos-errno.c, 189 pennos-errno.h, 194	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_PIPELINE, 243 pcb_st, 12 child_pcbs, 13 cmd_str, 13 fd_table, 13 input_fd, 13 is_sleeping, 13 output_fd, 14 par_pid, 14 pid, 14
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EPARSE pennos-errno.h, 192 P_EPERM pennos-errno.h, 192 P_EPERM pennos-errno.h, 193 P_EREAD pennos-errno.h, 193 P_ERRNO pennos-errno.c, 189 pennos-errno.h, 194 P_ESIGNAL	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_PIPELINE, 243 pcb_st, 12 child_pcbs, 13 cmd_str, 13 fd_table, 13 input_fd, 13 is_sleeping, 13 output_fd, 14 par_pid, 14 priority, 14
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EPARSE pennos-errno.h, 192 P_EPERM pennos-errno.h, 192 P_EREAD pennos-errno.h, 193 P_ERRNO pennos-errno.h, 194 P_ESIGNAL pennos-errno.h, 193	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_PIPELINE, 243 pcb_st, 12 child_pcbs, 13 cmd_str, 13 fd_table, 13 input_fd, 13 is_sleeping, 13 output_fd, 14 par_pid, 14 priority, 14 process_state, 14
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EPARSE pennos-errno.h, 192 P_EPERM pennos-errno.h, 192 P_EREAD pennos-errno.h, 193 P_ERRNO pennos-errno.h, 194 P_ESIGNAL pennos-errno.h, 193 P_EUNKNOWN	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_PIPELINE, 243 pcb_st, 12 child_pcbs, 13 cmd_str, 13 fd_table, 13 input_fd, 13 is_sleeping, 13 output_fd, 14 par_pid, 14 priority, 14
P_EINVAL pennos-errno.h, 191 P_ELSEEK pennos-errno.h, 191 P_EMALLOC pennos-errno.h, 191 P_EMAP pennos-errno.h, 192 P_ENOENT pennos-errno.h, 192 P_ENULL pennos-errno.h, 192 P_EOPEN pennos-errno.h, 192 P_EPARSE pennos-errno.h, 192 P_EPERM pennos-errno.h, 192 P_EREAD pennos-errno.h, 193 P_ERRNO pennos-errno.h, 194 P_ESIGNAL pennos-errno.h, 193	JUMP_OUT, 237 parse_command, 237 print_parsed_command, 240 print_parser_errcode, 240 parser.h EXPECT_COMMANDS, 242 EXPECT_INPUT_FILENAME, 242 EXPECT_OUTPUT_FILENAME, 243 parse_command, 244 print_parsed_command, 246 print_parser_errcode, 247 UNEXPECTED_AMPERSAND, 243 UNEXPECTED_FILE_INPUT, 243 UNEXPECTED_FILE_OUTPUT, 243 UNEXPECTED_PIPELINE, 243 pcb_st, 12 child_pcbs, 13 cmd_str, 13 fd_table, 13 input_fd, 13 is_sleeping, 13 output_fd, 14 par_pid, 14 priority, 14 process_state, 14

thread_handle, 15	pcb_st, 14
time_to_wake, 15	pids
pcb_t	job_st, 10
kern_pcb.h, 129	position
pennfat.c	fd_entry_t, 8
main, 227	print_parsed_command
PROMPT, 227	parser.c, 240
pennos-errno.c	parser.h, 246
P_ERRNO, 189	print_parser_errcode
pennos-errno.h	parser.c, 240
P_EBADF, 190	parser.h, 247
P_EBUSY, 190	priority
P_ECLOSE, 190	pcb_st, 14
P_ECOMMAND, 190	process_state
P_EEXIST, 190	pcb_st, 14
P_EFS_NOT_MOUNTED, 190 P_EFULL, 191	process_status
P_EPOLL, 191 P_EFUNC, 191	pcb_st, 14 PROMPT
P EINTR, 191	
P_EINTH, 191 P_EINVAL, 191	pennfat.c, 227
P_EINVAL, 191 P_ELSEEK, 191	shell.c, 249
P_ELSEEK, 191 P_EMALLOC, 191	pthread_fn spthread.c, 196
P_EMALLOG, 191 P_EMAP, 192	•
P ENOENT, 192	ptr_dtor_fn Vec.h, 219
P ENULL, 192	
P EOPEN, 192	ptr_t Vec.h, 219
P EPARSE, 192	put_pcb_into_correct_queue
P EPERM, 192	scheduler.c, 168
P EREAD, 193	scheduler.h, 180
P ERRNO, 194	Scheduler.ii, 180
	recur
P_ESIGNAL, 193	recur stress.c. 186
P_ESIGNAL, 193 P_EUNKNOWN, 193	stress.c, 186
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193	stress.c, 186 stress.h, 188
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193	stress.c, 186 stress.h, 188 ref_count
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39 PERM_READ	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39 PERM_READ fat_routines.h, 39	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING Job.h, 236 s_cleanup_init_process kern_sys_calls.c, 137
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ_EXEC	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING Job.h, 236 s_cleanup_init_process kern_sys_calls.c, 137 kern_sys_calls.h, 148
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39 PERM_READ fat_routines.h, 39	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING Job.h, 236 s_cleanup_init_process kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_close
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ_fat_routines.h, 39 PERM_READ_EXEC fat_routines.h, 39	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING Job.h, 236 s_cleanup_init_process kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_close fs_syscalls.c, 114
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ_tat_routines.h, 39 PERM_READ_EXEC fat_routines.h, 39 PERM_READ_EXEC fat_routines.h, 39 PERM_READ_WRITE fat_routines.h, 40	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING Job.h, 236 s_cleanup_init_process kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_close fs_syscalls.c, 114 fs_syscalls.h, 118
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ_EXEC fat_routines.h, 39 PERM_READ_WRITE fat_routines.h, 40 PERM_READ_WRITE_EXEC	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING Job.h, 236 s_cleanup_init_process kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_close fs_syscalls.c, 114 fs_syscalls.h, 118 s_echo
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ_EXEC fat_routines.h, 39 PERM_READ_WRITE fat_routines.h, 40 PERM_READ_WRITE_EXEC fat_routines.h, 40	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING Job.h, 236 s_cleanup_init_process kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_close fs_syscalls.c, 114 fs_syscalls.h, 118 s_echo kern_sys_calls.c, 137
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ_EXEC fat_routines.h, 39 PERM_READ_WRITE fat_routines.h, 40 PERM_READ_WRITE_EXEC fat_routines.h, 40 PERM_WRITE	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING Job.h, 236 s_cleanup_init_process kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_close fs_syscalls.c, 114 fs_syscalls.c, 114 fs_syscalls.c, 137 kern_sys_calls.c, 137 kern_sys_calls.c, 137 kern_sys_calls.c, 137 kern_sys_calls.c, 137 kern_sys_calls.c, 137 kern_sys_calls.h, 148
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ_EXEC fat_routines.h, 39 PERM_READ_WRITE fat_routines.h, 40 PERM_READ_WRITE_EXEC fat_routines.h, 40 PERM_WRITE fat_routines.h, 40 PERM_WRITE fat_routines.h, 40	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING Job.h, 236 s_cleanup_init_process kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_close fs_syscalls.c, 114 fs_syscalls.h, 118 s_echo kern_sys_calls.c, 137 kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_echo
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ_EXEC fat_routines.h, 39 PERM_READ_WRITE fat_routines.h, 40 PERM_READ_WRITE fat_routines.h, 40 PERM_WRITE fat_routines.h, 40 PERM_WRITE fat_routines.h, 40 pgid	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING Job.h, 236 s_cleanup_init_process kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_close fs_syscalls.c, 114 fs_syscalls.h, 118 s_echo kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_exit kern_sys_calls.c, 137
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ_EXEC fat_routines.h, 39 PERM_READ_EXEC fat_routines.h, 39 PERM_READ_WRITE fat_routines.h, 40 PERM_READ_WRITE_EXEC fat_routines.h, 40 PERM_WRITE fat_routines.h, 40 PERM_WRITE fat_routines.h, 40 pgid job_st, 9	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING Job.h, 236 s_cleanup_init_process kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_close fs_syscalls.c, 114 fs_syscalls.h, 118 s_echo kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_exit kern_sys_calls.c, 138 kern_sys_calls.h, 149
P_ESIGNAL, 193 P_EUNKNOWN, 193 P_EWRITE, 193 P_INITFAIL, 193 P_NEEDF, 193 pennos.c log_fd, 230 main, 230 tick_counter, 230 perm dir_entry_t, 6 PERM_EXEC fat_routines.h, 39 PERM_NONE fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ fat_routines.h, 39 PERM_READ_EXEC fat_routines.h, 39 PERM_READ_WRITE fat_routines.h, 40 PERM_READ_WRITE fat_routines.h, 40 PERM_WRITE fat_routines.h, 40 PERM_WRITE fat_routines.h, 40 pgid	stress.c, 186 stress.h, 188 ref_count fd_entry_t, 8 remove_child_in_parent kern_pcb.c, 126 kern_pcb.h, 133 reserved dir_entry_t, 6 rm fat_routines.c, 34 fat_routines.h, 52 RUNNING Job.h, 236 s_cleanup_init_process kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_close fs_syscalls.c, 114 fs_syscalls.h, 118 s_echo kern_sys_calls.c, 137 kern_sys_calls.h, 148 s_exit kern_sys_calls.c, 137

kern_sys_calls.c, 138	get_next_pcb, 166
kern_sys_calls.h, 149	get_pcb_in_queue, 166
s_ls	handle_signal, 167
fs_syscalls.c, 115	initialize_scheduler_queues, 167
fs_syscalls.h, 119	log_fd, 171
s_lseek	one_priority_queue, 171
fs_syscalls.c, 115	put_pcb_into_correct_queue, 168
fs_syscalls.h, 119	s_shutdown_pennos, 168
s_nice	scheduler, 168
kern_sys_calls.c, 138	sleep_blocked_queue, 171
kern_sys_calls.h, 151	tick_counter, 171
s_open	two_priority_queue, 171
fs_syscalls.c, 115	zero_priority_queue, 172
fs_syscalls.h, 120	zombie_queue, 172
s_ps	scheduler.h
kern_sys_calls.c, 139	alarm_handler, 173
kern_sys_calls.h, 151	child_in_zombie_queue, 174
s read	child with changed process status, 174
fs_syscalls.c, 116	delete_process_from_all_queues, 175
fs syscalls.h, 121	delete_process_from_all_queues_except_current,
s_shutdown_pennos	175
scheduler.c, 168	delete_process_from_particular_queue, 176
scheduler.h, 180	free_scheduler_queues, 176
s_sleep	generate_next_priority, 176
kern_sys_calls.c, 139	get_next_pcb, 177
kern_sys_calls.h, 152	get_noxt_pos, 177 get_pcb_in_queue, 178
— • —	handle_signal, 178
s_spawn kern_sys_calls.c, 140	initialize_scheduler_queues, 179
kern_sys_calls.b, 153	put_pcb_into_correct_queue, 180
— • —	. — — — — .
s_spawn_init	s_shutdown_pennos, 180
kern_sys_calls.c, 140	scheduler, 180
kern_sys_calls.h, 153	script_fd
s_unlink	shell.c, 257
fs_syscalls.c, 116	SEEK_CUR
fs_syscalls.h, 122	fs_kfuncs.h, 101
s_waitpid	SEEK_END
kern_sys_calls.c, 141	fs_kfuncs.h, 101
kern_sys_calls.h, 154	SEEK_SET
s_write	fs_kfuncs.h, 101
fs_syscalls.c, 116	setup_cond
fs_syscalls.h, 122	spthread_fwd_args_st, 16
scheduler	setup_done
scheduler.c, 168	spthread_fwd_args_st, 16
scheduler.h, 180	setup_mutex
scheduler.c	spthread_fwd_args_st, 17
alarm_handler, 163	setup_terminal_signal_handlers
child_in_zombie_queue, 163	shell.c, 251
child_with_changed_process_status, 164	shell
curr_priority_arr_index, 170	shell.c, 252
current_pcbs, 170	shell.h, 257
current_running_pcb, 170	shell.c
delete_process_from_all_queues, 164	current_fg_pid, 256
delete_process_from_all_queues_except_current,	execute_command, 249
164	fill_buffer_until_full_or_newline, 251
delete_process_from_particular_queue, 165	free_job_ptr, 251
det_priorities_arr, 171	input_fd_script, 256
free_scheduler_queues, 165	job_list, 256
generate_next_priority, 165	MAX_BUFFER_SIZE, 248
, ,·	

MAX_LINE_BUFFER_SIZE, 249	u_touch, 279
next_job_id, 256	u_zombify, 280
output_fd_script, 256	shell_sigint_handler
PROMPT, 249	shell.c, 253
script_fd, 257	shell_sigstp_handler
setup terminal signal handlers, 251	shell.c, 254
shell, 252	shutup_mutex
shell_sigint_handler, 253	spthread_signal_args_st, 18
shell_sigstp_handler, 254	signal
u_execute_command, 254	spthread_signal_args_st, 19
u_read_and_execute_script, 255	signal.h
shell.h	CONT_BY_SIG, 183
shell, 257	EXITED NORMALLY, 183
shell_built_ins.c	P_SIGCONT, 183
get_associated_ufunc, 269	P SIGSTOP, 184
-	P SIGTERM, 184
orphan_child, 261	-
u_bg, 261	P_WIFEXITED, 184
u_busy, 261	P_WIFSIGNALED, 184
u_cat, 261	P_WIFSTOPPED, 184
u_chmod, 262	STOPPED_BY_SIG, 185
u_cp, 262	TERM_BY_SIG, 185
u_echo, 263	signals
u_fg, 263	pcb_st, 15
u_jobs, 263	SIGPTHD
u_kill, 264	spthread.h, 203
	size
u_logout, 264	
u_ls, 265	dir_entry_t, 6
u_man, 265	fd_entry_t, 8
u_mv, 266	sleep_blocked_queue
u_nice, 266	kern_sys_calls.c, 143
u_nice_pid, 267	scheduler.c, 171
u_orphanify, 267	spthread.c
u_ps, 267	_GNU_SOURCE, 195
u rm, 268	MILISEC IN NANO, 195
u_sleep, 268	pthread_fn, 196
u_touch, 268	spthread_cancel, 197
	-
u_zombify, 269	spthread_continue, 197
zombie_child, 269	spthread_create, 198
shell_built_ins.h	spthread_disable_interrupts_self, 199
u_bg, 271	spthread_enable_interrupts_self, 199
u_busy, 272	spthread_equal, 199
u_cat, <mark>272</mark>	spthread_exit, 200
u_chmod, 272	spthread_fwd_args, 196
u_cp, 273	spthread_join, 200
u_echo, 273	spthread_meta_t, 197
u_fg, 274	SPTHREAD_RUNNING_STATE, 195
u_jobs, 274	spthread_self, 200
_ -	•
u_kill, 274	SPTHREAD_SIG_CONTINUE, 196
u_logout, 275	SPTHREAD_SIG_SUSPEND, 196
u_ls, 275	spthread_signal_args, 197
u_man, 276	spthread_suspend, 200
u_mv, 276	spthread_suspend_self, 201
u_nice, 277	SPTHREAD_SUSPENDED_STATE, 196
u_nice_pid, 277	SPTHREAD_TERMINATED_STATE, 196
u_orphanify, 278	spthread.h
u_ps, 278	SIGPTHD, 203
u_rm, 278	spthread_cancel, 203
u_sleep, 279	spthread_continue, 204
a_0000p, 270	optimoda_continuo, 204

spthread_create, 204	SPTHREAD_SIG_SUSPEND
spthread_disable_interrupts_self, 205	spthread.c, 196
spthread_enable_interrupts_self, 206	spthread_signal_args
spthread_equal, 206	spthread.c, 197
spthread_exit, 206	spthread_signal_args_st, 18
spthread_join, 206	ack, 18
spthread_meta_t, 203	shutup_mutex, 18
spthread_self, 207	signal, 19
spthread_suspend, 207	spthread st, 19
spthread_suspend_self, 208	meta, 20
spthread_t, 203	thread, 20
spthread_cancel	spthread_suspend
spthread.c, 197	spthread.c, 200
spthread.h, 203	spthread.h, 207
spthread_continue	spthread_suspend_self
spthread.c, 197	spthread.c, 201
spthread.h, 204	spthread.h, 208
spthread_create	SPTHREAD_SUSPENDED_STATE
spthread.c, 198	spthread.c, 196
spthread.h, 204	spthread_t
•	spthread_t spthread.h, 203
spthread_disable_interrupts_self	•
spthread.c, 199	SPTHREAD_TERMINATED_STATE
spthread.h, 205	spthread.c, 196
spthread_enable_interrupts_self	SRC/fs/fat_routines.c, 23
spthread.c, 199	SRC/fs/fat_routines.h, 36
spthread.h, 206	SRC/fs/fs_helpers.c, 55
spthread_equal	SRC/fs/fs_helpers.h, 71
spthread.c, 199	SRC/fs/fs_kfuncs.c, 88
spthread.h, 206	SRC/fs/fs_kfuncs.h, 100
spthread_exit	SRC/fs/fs_syscalls.c, 113
spthread.c, 200	SRC/fs/fs_syscalls.h, 117
spthread.h, 206	SRC/kernel/kern_pcb.c, 123
spthread_fwd_args	SRC/kernel/kern_pcb.h, 128
spthread.c, 196	SRC/kernel/kern_sys_calls.c, 133
spthread_fwd_args_st, 15	SRC/kernel/kern_sys_calls.h, 144
actual_arg, 16	SRC/kernel/logger.c, 155
actual_routine, 16	SRC/kernel/logger.h, 158
child_meta, 16	SRC/kernel/scheduler.c, 162
setup_cond, 16	SRC/kernel/scheduler.h, 172
setup_done, 16	SRC/kernel/signal.c, 182
setup_mutex, 17	SRC/kernel/signal.h, 183
spthread_join	SRC/kernel/stress.c, 185
spthread.c, 200	SRC/kernel/stress.h, 187
spthread.h, 206	SRC/lib/pennos-errno.c, 188
spthread_meta_st, 17	SRC/lib/pennos-errno.h, 189
meta mutex, 17	SRC/lib/spthread.c, 194
state, 17	SRC/lib/spthread.h, 202
suspend_set, 18	SRC/lib/Vec.c, 208
spthread_meta_t	SRC/lib/Vec.h, 216
spthread.c, 197	SRC/pennfat.c, 226
spthread.h, 203	SRC/pennos.c, 229
SPTHREAD_RUNNING_STATE	SRC/shell/builtins.c, 231
spthread.c, 195	SRC/shell/builtins.h, 233
spthread_self	SRC/shell/Job.h, 234
spthread_sell spthread.c, 200	SRC/shell/parser.c, 236
•	•
spthread.h, 207	SRC/shell/parser.h, 241
SPTHREAD_SIG_CONTINUE	SRC/shell/shell.c, 247
spthread.c, 196	SRC/shell/shell.h, 257

SRC/shell/shell_built_ins.c, 259	TYPE_UNKNOWN
SRC/shell/shell_built_ins.h, 270	fat_routines.h, 41
state	u ba
job_st, 10	u_bg
spthread_meta_st, 17	shell_built_ins.c, 261 shell_built_ins.h, 271
STDERR_FILENO	u_busy
fs_syscalls.h, 118	shell_built_ins.c, 261
stdin_file	shell_built_ins.h, 272
parsed_command, 11	u cat
STDIN_FILENO	shell_built_ins.c, 261
fs_syscalls.h, 118	shell_built_ins.h, 272
stdout_file	u chmod
parsed_command, 11 STDOUT FILENO	shell_built_ins.c, 262
fs_syscalls.h, 118	shell_built_ins.h, 272
STOPPED	u_cp
Job.h, 236	shell_built_ins.c, 262
STOPPED_BY_SIG	shell_built_ins.h, 273
signal.h, 185	u_echo
stress.c	shell_built_ins.c, 263
crash, 186	shell_built_ins.h, 273
hang, 186	u_execute_command
nohang, 186	shell.c, 254
recur, 186	u_fg
stress.h	shell_built_ins.c, 263
crash, 187	shell_built_ins.h, 274
hang, 187	u_jobs
nohang, 188	shell_built_ins.c, 263
recur, 188	shell_built_ins.h, 274
suspend_set	u_kill
spthread_meta_st, 18	shell_built_ins.c, 264
TERM BY 616	shell_built_ins.h, 274
TERM_BY_SIG	u_logout
signal.h, 185	shell_built_ins.c, 264
thread st, 20	shell_built_ins.h, 275
thread_handle	u_ls
pcb_st, 15	shell_built_ins.c, 265 shell_built_ins.h, 275
tick counter	u man
kern_sys_calls.c, 143	shell_built_ins.c, 265
logger.h, 161	shell_built_ins.h, 276
pennos.c, 230	u_mv
scheduler.c, 171	shell_built_ins.c, 266
time_to_wake	shell_built_ins.h, 276
 pcb_st, 15	u_nice
touch	shell_built_ins.c, 266
fat_routines.c, 35	shell_built_ins.h, 277
fat_routines.h, 53	u_nice_pid
two_priority_queue	shell_built_ins.c, 267
kern_sys_calls.c, 143	shell_built_ins.h, 277
scheduler.c, 171	u_orphanify
type	shell_built_ins.c, 267
dir_entry_t, 6	shell_built_ins.h, 278
TYPE_DIRECTORY	u_perror
fat_routines.h, 40	builtins.c, 231
TYPE_REGULAR	builtins.h, 233
fat_routines.h, 40	u_ps
TYPE_SYMLINK	shell_built_ins.c, 267
fat_routines.h, 40	shell_built_ins.h, 278

	V 1 040
u_read_and_execute_script	Vec.h, 218
shell.c, 255	vec_clear
u_rm	Vec.c, 209
shell_built_ins.c, 268	Vec.h, 219
shell_built_ins.h, 278	vec destroy
u_sleep	Vec.c, 210
shell_built_ins.c, 268	Vec.h, 220
shell_built_ins.h, 279	vec_erase
u_touch	Vec.c, 210
shell_built_ins.c, 268	Vec.h, 220
shell_built_ins.h, 279	vec_erase_no_deletor
u zombify	Vec.c, 211
shell_built_ins.c, 269	Vec.h, 221
shell built ins.h, 280	vec get
UNEXPECTED_AMPERSAND	Vec.c, 211
parser.h, 243	Vec.h, 221
UNEXPECTED_FILE_INPUT	vec_insert
parser.h, 243	Vec.c, 212
UNEXPECTED FILE OUTPUT	Vec.h, 222
parser.h, 243	vec is empty
UNEXPECTED PIPELINE	' '
-	Vec.h, 218
parser.h, 243	vec_len
unmount	Vec.h, 218
fat_routines.c, 36	vec_new
fat routines.h, 54	Vec.c, 213
	Vec.h, 223
Vec	vec_pop_back
Vec.h, 219	_,
Vec.c	Vec.c, 213
	Vec.h, 223
vec_clear, 209	vec_push_back
vec_destroy, 210	Vec.c, 214
vec_erase, 210	Vec.h, 224
vec erase no deletor, 211	vec_resize
vec_get, 211	
vec_insert, 212	Vec.c, 215
vec new, 213	Vec.h, 225
— · · · · · · · · · · · · · · · · · · ·	vec_set
vec_pop_back, 213	Vec.c, 216
vec_push_back, 214	Vec.h, 226
vec_resize, 215	vec_st, 20
vec set, 216	capacity, 20
Vec.h	•
ptr_dtor_fn, 219	data, 21
• — —	ele_dtor_fn, 21
ptr_t, 219	length, 21
Vec, 219	
vec_capacity, 218	zero_priority_queue
vec_clear, 219	
	kern sys calls.c, 143
vec destroy, 220	kern_sys_calls.c, 143 scheduler.c. 172
vec_destroy, 220	scheduler.c, 172
vec_erase, 220	scheduler.c, 172 zombie_child
vec_erase_no_deletor, 221	scheduler.c, 172 zombie_child shell_built_ins.c, 269
vec_erase, 220 vec_erase_no_deletor, 221 vec_get, 221	scheduler.c, 172 zombie_child shell_built_ins.c, 269 zombie_queue
vec_erase, 220 vec_erase_no_deletor, 221 vec_get, 221 vec_insert, 222	scheduler.c, 172 zombie_child shell_built_ins.c, 269 zombie_queue kern_sys_calls.c, 143
vec_erase, 220 vec_erase_no_deletor, 221 vec_get, 221 vec_insert, 222 vec_is_empty, 218	scheduler.c, 172 zombie_child shell_built_ins.c, 269 zombie_queue
vec_erase, 220 vec_erase_no_deletor, 221 vec_get, 221 vec_insert, 222	scheduler.c, 172 zombie_child shell_built_ins.c, 269 zombie_queue kern_sys_calls.c, 143
vec_erase, 220 vec_erase_no_deletor, 221 vec_get, 221 vec_insert, 222 vec_is_empty, 218	scheduler.c, 172 zombie_child shell_built_ins.c, 269 zombie_queue kern_sys_calls.c, 143
vec_erase, 220 vec_erase_no_deletor, 221 vec_get, 221 vec_insert, 222 vec_is_empty, 218 vec_len, 218 vec_new, 223	scheduler.c, 172 zombie_child shell_built_ins.c, 269 zombie_queue kern_sys_calls.c, 143
vec_erase, 220 vec_erase_no_deletor, 221 vec_get, 221 vec_insert, 222 vec_is_empty, 218 vec_len, 218 vec_new, 223 vec_pop_back, 223	scheduler.c, 172 zombie_child shell_built_ins.c, 269 zombie_queue kern_sys_calls.c, 143
vec_erase, 220 vec_erase_no_deletor, 221 vec_get, 221 vec_insert, 222 vec_is_empty, 218 vec_len, 218 vec_new, 223 vec_pop_back, 223 vec_push_back, 224	scheduler.c, 172 zombie_child shell_built_ins.c, 269 zombie_queue kern_sys_calls.c, 143
vec_erase, 220 vec_erase_no_deletor, 221 vec_get, 221 vec_insert, 222 vec_is_empty, 218 vec_len, 218 vec_new, 223 vec_pop_back, 223 vec_push_back, 224 vec_resize, 225	scheduler.c, 172 zombie_child shell_built_ins.c, 269 zombie_queue kern_sys_calls.c, 143
vec_erase, 220 vec_erase_no_deletor, 221 vec_get, 221 vec_insert, 222 vec_is_empty, 218 vec_len, 218 vec_new, 223 vec_pop_back, 223 vec_push_back, 224 vec_resize, 225 vec_set, 226	scheduler.c, 172 zombie_child shell_built_ins.c, 269 zombie_queue kern_sys_calls.c, 143
vec_erase, 220 vec_erase_no_deletor, 221 vec_get, 221 vec_insert, 222 vec_is_empty, 218 vec_len, 218 vec_new, 223 vec_pop_back, 223 vec_push_back, 224 vec_resize, 225	scheduler.c, 172 zombie_child shell_built_ins.c, 269 zombie_queue kern_sys_calls.c, 143