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	
4.1.1 Function Documentation	
4.1.1.1 cat()	
4.1.1.2 chmod()	
4.1.1.3 cmpctdir()	
4.1.1.4 cp()	
4.1.1.4 cp()	
4.1.1.6 mkfs()	
4.1.1.8 mv()	
4.1.1.9 rm()	
4.1.1.10 touch()	
4.1.1.11 unmount()	
4.1.2 Variable Documentation	
4.1.2.1 current_running_pcb	
4.2 SRC/fs/fat_routines.h File Reference	
4.2.1 Macro Definition Documentation	
4.2.1.1 F_APPEND	
4.2.1.2 F_READ	
4.2.1.3 F_WRITE	
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	40
4.2.1.8 PERM_READ	40
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	. 41
4.2.1.14 TYPE_REGULAR	. 41
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()	. 45
4.2.2.3 cmpctdir()	. 46
4.2.2.4 cp()	. 47
4.2.2.5 ls()	. 48
4.2.2.6 mkfs()	. 49
4.2.2.7 mount()	. 50
4.2.2.8 mv()	. 51
4.2.2.9 rm()	. 53
4.2.2.10 touch()	. 54
4.2.2.11 unmount()	. 55
4.3 SRC/fs/fs_helpers.c File Reference	. 56
4.3.1 Function Documentation	. 57
4.3.1.1 add_file_entry()	. 57
4.3.1.2 allocate_block()	. 59
4.3.1.3 compact_directory()	. 59
4.3.1.4 copy_host_to_pennfat()	. 62
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()	. 68
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 k_close()	89
4.5.1.2 k_ls()	90
4.5.1.3 k_lseek()	92
4.5.1.4 k_open()	92
4.5.1.5 k_read()	94
4.5.1.6 k_unlink()	96
4.5.1.7 k_write()	97
4.5.2 Variable Documentation	100
4.5.2.1 current_fg_pid	100
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	102
4.6.2.1 k_close()	102
4.6.2.2 k_ls()	103

4.6.2.3 k_lseek()
4.6.2.4 k_open()
4.6.2.5 k_read()
4.6.2.6 k_unlink()
4.6.2.7 k_write()
4.7 SRC/fs/fs_syscalls.c File Reference
4.7.1 Function Documentation
4.7.1.1 s_close()
4.7.1.2 s_ls()
4.7.1.3 s_lseek()
4.7.1.4 s_open()
4.7.1.5 s_read()
4.7.1.6 s_unlink()
4.7.1.7 s_write()
4.8 SRC/fs/fs_syscalls.h File Reference
4.8.1 Macro Definition Documentation
4.8.1.1 STDERR_FILENO
4.8.1.2 STDIN_FILENO
4.8.1.3 STDOUT_FILENO
4.8.2 Function Documentation
4.8.2.1 s_close()
4.8.2.2 s_ls()
4.8.2.3 s_lseek()
4.8.2.4 s_open()
4.8.2.5 s_read()
4.8.2.6 s_unlink()
4.8.2.7 s_write()
4.9 SRC/kernel/kern_pcb.c File Reference
4.9.1 Function Documentation
4.9.1.1 create_pcb()
4.9.1.2 free_pcb()
4.9.1.3 k_proc_cleanup()
4.9.1.4 k_proc_create()
4.9.1.5 remove_child_in_parent()
4.9.2 Variable Documentation
4.9.2.1 current_pcbs
4.9.2.2 current_running_pcb
4.9.2.3 next_pid
4.10 SRC/kernel/kern_pcb.h File Reference
4.10.1 Macro Definition Documentation
4.10.1.1 FILE_DESCRIPTOR_TABLE_SIZE
4 10.2 Typedef Documentation

4.10.2.1 pcb_t	30
4.10.3 Function Documentation	30
4.10.3.1 create_pcb()	30
4.10.3.2 free_pcb()	31
4.10.3.3 k_proc_cleanup()	32
4.10.3.4 k_proc_create()	33
4.10.3.5 remove_child_in_parent()	34
4.11 SRC/kernel/kern_sys_calls.c File Reference	34
4.11.1 Function Documentation	36
4.11.1.1 delete_from_explicit_queue()	36
4.11.1.2 delete_from_queue()	36
4.11.1.3 determine_index_in_queue()	37
4.11.1.4 init_func()	37
4.11.1.5 move_pcb_correct_queue()	37
4.11.1.6 s_cleanup_init_process()	38
4.11.1.7 s_echo()	38
4.11.1.8 s_exit()	39
4.11.1.9 s_kill()	39
4.11.1.10 s_nice()	40
4.11.1.11 s_ps()	40
4.11.1.12 s_sleep()	41
4.11.1.13 s_spawn()	41
4.11.1.14 s_spawn_init()	42
4.11.1.15 s_waitpid()	42
4.11.2 Variable Documentation	43
4.11.2.1 current_fg_pid	43
4.11.2.2 current_pcbs	43
4.11.2.3 current_running_pcb	44
4.11.2.4 one_priority_queue	44
4.11.2.5 sleep_blocked_queue	44
4.11.2.6 tick_counter	44
4.11.2.7 two_priority_queue	44
4.11.2.8 zero_priority_queue	44
4.11.2.9 zombie_queue	45
4.12 SRC/kernel/kern_sys_calls.h File Reference	45
4.12.1 Function Documentation	46
4.12.1.1 delete_from_explicit_queue()	46
4.12.1.2 delete_from_queue()	47
4.12.1.3 determine_index_in_queue()	47
4.12.1.4 init_func()	48
4.12.1.5 move_pcb_correct_queue()	48
4.12.1.6 s_cleanup_init_process()	49

4.12.1.7 s_echo()
4.12.1.8 s_exit()
4.12.1.9 s_kill()
4.12.1.10 s_nice()
4.12.1.11 s_ps()
4.12.1.12 s_sleep()
4.12.1.13 s_spawn()
4.12.1.14 s_spawn_init()
4.12.1.15 s_waitpid()
4.13 SRC/kernel/logger.c File Reference
4.13.1 Function Documentation
4.13.1.1 log_generic_event()
4.13.1.2 log_nice_event()
4.13.1.3 log_scheduling_event()
4.14 SRC/kernel/logger.h File Reference
4.14.1 Function Documentation
4.14.1.1 log_generic_event()
4.14.1.2 log_nice_event()
4.14.1.3 log_scheduling_event()
4.14.2 Variable Documentation
4.14.2.1 log_fd
4.14.2.2 tick_counter
4.15 SRC/kernel/scheduler.c File Reference
4.15.1 Function Documentation
4.15.1.1 alarm_handler()
4.15.1.2 child_in_zombie_queue()
4.15.1.3 child_with_changed_process_status()
4.15.1.4 delete_process_from_all_queues()
4.15.1.5 delete_process_from_all_queues_except_current() 165
4.15.1.6 delete_process_from_particular_queue()
4.15.1.7 free_scheduler_queues()
4.15.1.8 generate_next_priority()
4.15.1.9 get_next_pcb()
4.15.1.10 get_pcb_in_queue()
4.15.1.11 handle_signal()
4.15.1.12 initialize_scheduler_queues()
4.15.1.13 put_pcb_into_correct_queue()
4.15.1.14 s_shutdown_pennos()
4.15.1.15 scheduler()
4.15.2 Variable Documentation
4.15.2.1 curr_priority_arr_index
4.15.2.2 current_pcbs

4.15.2.3 current_running_pcb	171
4.15.2.4 det_priorities_arr	171
4.15.2.5 log_fd	171
4.15.2.6 one_priority_queue	171
4.15.2.7 sleep_blocked_queue	171
4.15.2.8 tick_counter	171
4.15.2.9 two_priority_queue	172
4.15.2.10 zero_priority_queue	172
4.15.2.11 zombie_queue	172
4.16 SRC/kernel/scheduler.h File Reference	172
4.16.1 Function Documentation	173
4.16.1.1 alarm_handler()	173
4.16.1.2 child_in_zombie_queue()	174
4.16.1.3 child_with_changed_process_status()	174
4.16.1.4 delete_process_from_all_queues()	175
4.16.1.5 delete_process_from_all_queues_except_current()	175
4.16.1.6 delete_process_from_particular_queue()	176
4.16.1.7 free_scheduler_queues()	176
4.16.1.8 generate_next_priority()	177
4.16.1.9 get_next_pcb()	177
4.16.1.10 get_pcb_in_queue()	178
4.16.1.11 handle_signal()	179
4.16.1.12 initialize_scheduler_queues()	179
4.16.1.13 put_pcb_into_correct_queue()	180
4.16.1.14 s_shutdown_pennos()	180
4.16.1.15 scheduler()	181
4.17 SRC/kernel/signal.c File Reference	182
4.18 SRC/kernel/signal.h File Reference	183
4.18.1 Macro Definition Documentation	183
4.18.1.1 CONT_BY_SIG	183
4.18.1.2 EXITED_NORMALLY	183
4.18.1.3 P_SIGCONT	184
4.18.1.4 P_SIGSTOP	184
4.18.1.5 P_SIGTERM	184
4.18.1.6 P_WIFEXITED	184
4.18.1.7 P_WIFSIGNALED	184
4.18.1.8 P_WIFSTOPPED	185
4.18.1.9 STOPPED_BY_SIG	185
4.18.1.10 TERM_BY_SIG	185
4.19 SRC/kernel/stress.c File Reference	185
4.19.1 Function Documentation	186
4.19.1.1 crash()	186

4.19.1.2 hang()
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_EREDIR
4.22.1.21 P_ESIGNAL
4.22.1.22 P_EUNKNOWN
4.22.1.23 P_EWRITE
4.22.1.24 P_INITFAIL
4.22.1.25 P_NEEDF
4.22.2 Variable Documentation
4.22.2.1 P_ERRNO
4.23 SBC/lib/spthread.c File Reference 194

4.23.1 Macro Definition Documentation	 195
4.23.1.1 _GNU_SOURCE	 195
4.23.1.2 MILISEC_IN_NANO	 196
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	 197
4.23.2.1 pthread_fn	 197
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()	 198
4.23.3.3 spthread_create()	 198
4.23.3.4 spthread_disable_interrupts_self()	 199
4.23.3.5 spthread_enable_interrupts_self()	 200
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()	 201
4.23.3.10 spthread_suspend()	 201
4.23.3.11 spthread_suspend_self()	 202
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	 204
4.24.3 Function Documentation	 204
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()	 206
4.24.3.5 spthread_enable_interrupts_self()	
4.24.3.6 spthread_equal()	 206
4.24.3.7 spthread_exit()	 207
4.24.3.8 spthread_join()	 207
4.24.3.9 spthread_self()	 207
4.24.3.10 spthread_suspend()	

4.24.3.11 spthread_suspend_self()	208
4.25 SRC/lib/Vec.c File Reference	209
4.25.1 Function Documentation	209
4.25.1.1 vec_clear()	209
4.25.1.2 vec_destroy()	210
4.25.1.3 vec_erase()	211
4.25.1.4 vec_erase_no_deletor()	211
4.25.1.5 vec_get()	212
4.25.1.6 vec_insert()	212
4.25.1.7 vec_new()	213
4.25.1.8 vec_pop_back()	214
4.25.1.9 vec_push_back()	214
4.25.1.10 vec_resize()	215
4.25.1.11 vec_set()	216
4.26 SRC/lib/Vec.h File Reference	217
4.26.1 Macro Definition Documentation	218
4.26.1.1 vec_capacity	218
4.26.1.2 vec_is_empty	218
4.26.1.3 vec_len	219
4.26.2 Typedef Documentation	219
4.26.2.1 ptr_dtor_fn	
4.26.2.2 ptr_t	219
4.26.2.3 Vec	219
4.26.3 Function Documentation	219
4.26.3.1 vec_clear()	219
4.26.3.2 vec_destroy()	220
4.26.3.3 vec_erase()	221
4.26.3.4 vec_erase_no_deletor()	221
4.26.3.5 vec_get()	222
4.26.3.6 vec_insert()	222
4.26.3.7 vec_new()	223
4.26.3.8 vec_pop_back()	224
4.26.3.9 vec_push_back()	224
4.26.3.10 vec_resize()	225
4.26.3.11 vec_set()	226
4.27 SRC/pennfat.c File Reference	227
4.27.1 Macro Definition Documentation	227
4.27.1.1 PROMPT	227
4.27.2 Function Documentation	227
4.27.2.1 main()	228
4.28 SRC/pennos.c File Reference	229
4.28.1 Function Documentation	230

4.28.1.1 main()	30
4.28.2 Variable Documentation	30
4.28.2.1 log_fd	30
4.28.2.2 tick_counter	31
4.29 SRC/shell/builtins.c File Reference	31
4.29.1 Function Documentation	31
4.29.1.1 u_perror()	31
4.30 SRC/shell/builtins.h File Reference	33
4.30.1 Function Documentation	33
4.30.1.1 u_perror()	33
4.31 SRC/shell/Job.h File Reference	
4.31.1 Typedef Documentation	35
4.31.1.1 jid_t	
4.31.1.2 job	36
4.31.2 Enumeration Type Documentation	36
4.31.2.1 job_state_t	36
4.32 SRC/shell/parser.c File Reference	36
4.32.1 Macro Definition Documentation	37
4.32.1.1 JUMP_OUT	37
4.32.2 Function Documentation	37
4.32.2.1 parse_command()	38
4.32.2.2 print_parsed_command()	40
4.32.2.3 print_parser_errcode()	41
4.33 SRC/shell/parser.h File Reference	41
4.33.1 Macro Definition Documentation	42
4.33.1.1 EXPECT_COMMANDS	42
4.33.1.2 EXPECT_INPUT_FILENAME	43
4.33.1.3 EXPECT_OUTPUT_FILENAME	
4.33.1.4 UNEXPECTED_AMPERSAND	43
4.33.1.5 UNEXPECTED_FILE_INPUT	43
4.33.1.6 UNEXPECTED_FILE_OUTPUT	43
4.33.1.7 UNEXPECTED_PIPELINE	43
4.33.2 Function Documentation	44
4.33.2.1 parse_command()	
4.33.2.2 print_parsed_command()	46
4.33.2.3 print_parser_errcode()	47
4.34 SRC/shell/shell.c File Reference	47
4.34.1 Macro Definition Documentation	
4.34.1.1 MAX_BUFFER_SIZE	
4.34.1.2 MAX_LINE_BUFFER_SIZE	49
4.34.1.3 PROMPT	49
4.34.2 Function Documentation	49

4.34.2.1 execute_command()	. 249
4.34.2.2 fill_buffer_until_full_or_newline()	. 251
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()	. 254
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 is_append	. 257
4.34.3.4 job_list	. 257
4.34.3.5 next_job_id	. 257
4.34.3.6 output_fd_script	. 257
4.34.3.7 script_fd	. 257
4.35 SRC/shell/shell.h File Reference	. 257
4.35.1 Function Documentation	. 258
4.35.1.1 shell()	. 258
4.35.2 Variable Documentation	. 259
4.35.2.1 is_append	. 259
4.36 SRC/shell/shell_built_ins.c File Reference	. 260
4.36.1 Function Documentation	. 261
4.36.1.1 orphan_child()	. 261
4.36.1.2 u_bg()	. 262
4.36.1.3 u_busy()	. 262
4.36.1.4 u_cat()	. 262
4.36.1.5 u_chmod()	. 263
4.36.1.6 u_cp()	. 263
4.36.1.7 u_echo()	. 264
4.36.1.8 u_fg()	. 264
4.36.1.9 u_jobs()	. 264
4.36.1.10 u_kill()	. 265
4.36.1.11 u_logout()	. 265
4.36.1.12 u_ls()	. 266
4.36.1.13 u_man()	. 266
4.36.1.14 u_mv()	. 267
4.36.1.15 u_nice()	. 267
4.36.1.16 u_nice_pid()	. 268
4.36.1.17 u_orphanify()	. 268
4.36.1.18 u_ps()	. 268

283

4.36.1.19 u_rm()	269
4.36.1.20 u_sleep()	269
4.36.1.21 u_touch()	270
4.36.1.22 u_zombify()	270
4.36.1.23 zombie_child()	270
4.36.2 Variable Documentation	270
4.36.2.1 get_associated_ufunc	270
4.37 SRC/shell/shell_built_ins.h File Reference	271
4.37.1 Function Documentation	272
4.37.1.1 u_bg()	273
4.37.1.2 u_busy()	273
4.37.1.3 u_cat()	273
4.37.1.4 u_chmod()	274
4.37.1.5 u_cp()	274
4.37.1.6 u_echo()	275
4.37.1.7 u_fg()	275
4.37.1.8 u_jobs()	275
4.37.1.9 u_kill()	276
4.37.1.10 u_logout()	276
4.37.1.11 u_ls()	277
4.37.1.12 u_man()	277
4.37.1.13 u_mv()	278
4.37.1.14 u_nice()	278
4.37.1.15 u_nice_pid()	279
4.37.1.16 u_orphanify()	279
4.37.1.17 u_ps()	279
4.37.1.18 u_rm()	280
4.37.1.19 u_sleep()	280
4.37.1.20 u_touch()	281
4.37.1.21 u_zombify()	281

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.

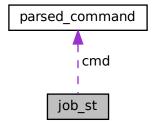
The documentation for this struct was generated from the following file:

• 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:

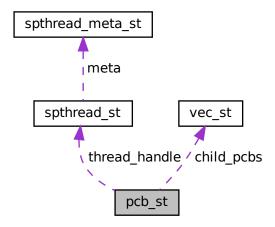
· SRC/shell/parser.h

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
```

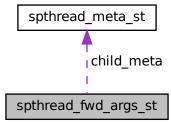
Definition at line 59 of file kern_pcb.h.

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.

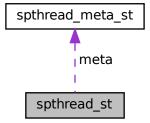
The documentation for this struct was generated from the following file:

• 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

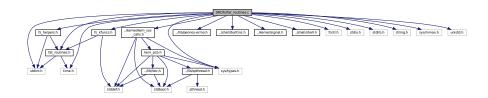
Chapter 4

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 "../kernel/kern_sys_calls.h"
#include "../kernel/signal.h"
#include "../shell/shell.h"
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
```

Include dependency graph for fat_routines.c:



Functions

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

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

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

• int unmount ()

Unmounts the current filesystem and reset variables.

void * cat (void *arg)

Concatenates and displays files.

void * 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.

Variables

• pcb_t * current_running_pcb

4.1.1 Function Documentation

4.1.1.1 cat()

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

Concatenates and displays files.

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

Usage formats:

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

Parameters

arg | Arguments array (command line arguments)

Returns

void pointer (unused)

```
Definition at line 214 of file fat_routines.c.
215
              char** args = (char**)arg;
216
             \ensuremath{//} verify that the file system is mounted
217
             if (!is_mounted) {
  P_ERRNO = P_EFS_NOT_MOUNTED;
218
219
220
                 u_perror("cat");
221
                 return NULL;
222
223
             // early return if there is nothing after cat
224
             if (args[1] == NULL) {
225
226
                // if none of the above conditions, then check if we need to redirect stdin
227
                  if (current_running_pcb) {
228
                       // open new stdin
                      int in_fd = current_running_pcb->input_fd;
229
                      int out_fd = current_running_pcb->output_fd;
char* file_1 = fd_table[in_fd].filename;
230
231
                      char* file_2 = fd_table[out_fd].filename;
233
234
                       // edge case when input and output have the same file name and \ensuremath{\mathsf{we'}}\xspace representation and the same file name and <math>\ensuremath{\mathsf{we'}}\xspace representation and the same file name and the same file name and <math>\ensuremath{\mathsf{we'}}\xspace representation and the same file name file name and the same file name and the same file name file name and the same file name file nam
235
                      if ((strcmp(file_1, file_2) == 0) && is_append) {
   P_ERRNO = P_EREDIR;
236
237
                          u_perror("cat");
238
                          return NULL;
239
240
                      // edge case when input and output files names are the same but we're not appending
241
242
                       // truncates the file
243
                      if ((strcmp(file_1, file_2) == 0)) {
                          return NULL;
245
246
247
                       \ensuremath{//} get the size of stdin file
                      off_t in_fd_size = k_lseek(in_fd, 0, SEEK_END);
2.48
                      if (in_fd_size == -1) {
249
250
                         k_close(in_fd);
                          u_perror("cat");
252
                          return NULL;
253
254
                       if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
255
                          k close(in fd);
                           u_perror("cat");
256
                          return NULL;
258
259
                      char* buffer = (char*)malloc(block size);
260
                      if (buffer == NULL) {
261
                         P_ERRNO = P_EMALLOC;
262
263
                           k_close(in_fd);
                          u_perror("cat");
264
265
                           return NULL;
266
2.67
268
                      int bytes read;
269
                      ssize_t bytes_remaining = in_fd_size;
270
271
                      while (bytes_remaining > 0) {
                          ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
272
273
                          bytes_read = k_read(in_fd, buffer, bytes_to_read);
274
275
                           if (bytes_read <= 0) {</pre>
276
                              break;
277
278
279
                           if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
280
                              free (buffer);
281
                                k close(in fd);
                                u_perror("cat");
283
                               break;
```

284

```
285
286
             bytes_remaining -= bytes_read;
287
288
           // read error
if (bytes_read < 0) {</pre>
289
290
291
            free (buffer);
292
              k_close(in_fd);
             u_perror("cat");
293
294
             return NULL;
           }
295
296
           k_close(in_fd);
if (out_fd != STDOUT_FILENO) {
297
298
299
             k_close(out_fd);
300
           free (buffer);
301
302
           return NULL;
303
         P_ERRNO = P_EINVAL;
u_perror("cat");
304
305
306
         return NULL;
307
308
309
       // check for output file with -w or -a flag
       int out_fd = -1;
310
311
       int out_mode = 0;
312
313
       \ensuremath{//} scan arguments and determine output fd and output mode
314
       int i;
       for (i = 1; args[i] != NULL; i++) {
315
316
         if (strcmp(args[i], "-w") == 0 && args[i + 1] != NULL) {
317
           out_mode = F_WRITE;
            out_fd = k_open(args[i + 1], F_WRITE);
318
           if (out_fd < 0) {
   u_perror("cat");</pre>
319
320
321
             return NULL;
322
323
           break;
324
         } else if (strcmp(args[i], "-a") == 0 && args[i + 1] != NULL) {
           out_mode = F_APPEND;
out_fd = k_open(args[i + 1], F_APPEND);
325
326
           if (out_fd < 0) {</pre>
327
            u_perror("cat");
return NULL;
328
329
330
331
           break;
332
         }
      }
333
334
335
       // if no output redirection found, use STDOUT
336
       if (out_fd < 0) {</pre>
337
        if (current_running_pcb) {
338
           out_fd = current_running_pcb->output_fd;
339
        } else {
           out_fd = STDOUT_FILENO;
340
341
342
343
      // handle small case: cat -w OUTPUT_FILE or cat -a OUTPUT_FILE (read from stdin) if ((strcmp(args[1], "-w") == 0 || strcmp(args[1], "-a") == 0) && args[2] != NULL && args[3] == NULL) {
344
345
346
347
         char buffer[1024];
348
349
         while (1) {
           ssize_t bytes_read = k_read(STDIN_FILENO, buffer, sizeof(buffer));
350
351
           if (bytes read < 0) {
352
             u_perror("cat");
if (out_fd != STDOUT_FILENO) {
353
354
355
               k_close(out_fd);
356
357
              return NULL;
358
359
360
            if (bytes_read == 0) {
361
             break;
362
363
364
            if (k write(out fd, buffer, bytes read) != bytes read) {
             u_perror("cat");
if (out_fd != STDOUT_FILENO) {
365
366
367
               k_close(out_fd);
              }
368
369
              return NULL;
370
           }
371
         }
```

```
373
         if (out_fd != STDOUT_FILENO) {
374
           k_close(out_fd);
         }
375
376
         return NULL;
377
378
379
       // handle concatenating one or more files: cat FILE ... [-w/-a OUTPUT_FILE]
       int start = 1;
380
381
       int end = i - 1;
382
      end = i - 1; // skip the output redirection arguments }
383
384
385
386
387
       // process each input file
       for (i = start; i <= end; i++) {</pre>
388
        // skip the redirection flags and their arguments
if (strcmp(args[i], "-w") == 0 || strcmp(args[i], "-a") == 0) {
389
390
391
           i++;
           continue;
392
393
394
         // open the current input file
int in_fd = k_open(args[i], F_READ);
if (in_fd < 0) {</pre>
395
396
397
           u_perror("cat");
398
399
           continue;
400
401
         // 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) {
402
403
404
405
           k_close(in_fd);
406
           u_perror("cat");
407
           continue;
408
409
410
         // use lseek to reset position to 0 for reading
411
         if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
412
           k_close(in_fd);
           u_perror("cat");
413
414
           continue;
415
416
417
         // copy file content to output
         char* buffer = (char*)malloc(block_size);
if (buffer == NULL) {
418
419
           P_ERRNO = P_EMALLOC;
420
421
           k_close(in_fd);
422
           u_perror("cat");
423
           continue;
424
425
         int bytes_read;
426
427
         ssize_t bytes_remaining = in_fd_size;
428
429
         while (bytes_remaining > 0) {
430
           ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
431
           bytes_read = k_read(in_fd, buffer, bytes_to_read);
432
433
           if (bytes_read <= 0) {</pre>
434
             break;
435
436
437
           if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
438
             free (buffer);
439
              k close(in fd);
440
             u_perror("cat");
441
             break;
442
443
444
           bytes_remaining -= bytes_read;
445
446
447
         // read error
448
         if (bytes_read < 0) {</pre>
449
           free (buffer);
           k_close(in_fd);
u_perror("cat");
450
451
452
           continue;
453
454
455
         k_close(in_fd);
456
        free(buffer);
457
458
```

```
459  // close output file if not stdout
460  if (out_fd != STDOUT_FILENO) {
461    k_close(out_fd);
462  }
463
464  return NULL;
465 }
```

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 765 of file fat_routines.c.

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

```
813
            default:
814
             P_ERRNO = P_EINVAL;
815
               return NULL;
816
         i++;
817
818
       }
819
820
       // Update the directory entry
       dir_entry.perm = new_perm;
dir_entry.mtime = time(NULL);
821
822
823
       // Seek to the entry's position
if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
824
825
826
827
          return NULL;
828
829
830
       // Write the updated entry back
       if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
   P_ERRNO = P_EWRITE;
   return NULL;
831
832
833
834
835
836
       return NULL;
837 }
```

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 846 of file fat_routines.c.

```
846
847
       if (!is_mounted) {
         P_ERRNO = P_EFS_NOT_MOUNTED;
u_perror("cmpctdir");
849
850
          return NULL;
851
852
       if (compact_directory() != 0) {
  u_perror("cmpctdir");
853
855 }
856
857
       return NULL;
858 }
```

4.1.1.4 cp()

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

Copies the source file to the destination.

Copies files.

Definition at line 635 of file fat_routines.c.

```
635 {
636 char** args = (char**)arg;
637
638 // check that we have enough arguments
639 if (args[1] == NULL || args[2] == NULL) {
```

```
P_ERRNO = P_EINVAL;
640
641
         u_perror("cp");
642
         return NULL;
643
644
645
      // cp -h SOURCE DEST
      if (strcmp(args[1], "-h") == 0) {
646
       if (args[2] == NULL || args[3] == NULL) {
   P_ERRNO = P_EINVAL;
   u_perror("cp");
647
648
649
          return NULL;
650
651
652
653
         if (copy_host_to_pennfat(args[2], args[3]) != 0) {
654
          u_perror("cp");
655
           return NULL;
656
         return NULL;
657
658
659
660
       // cp SOURCE -h DEST
661
       if (args[2] != NULL && strcmp(args[2], "-h") == 0) {
       if (args[3] == NULL) {
   P_ERRNO = P_EINVAL;
662
663
664
           u_perror("cp");
665
           return NULL;
666
667
        if (copy_pennfat_to_host(args[1], args[3]) != 0) {
  u_perror("cp");
668
669
670
          return NULL:
671
672
         return NULL;
673
      }
674
675
      // cp SOURCE DEST
      if ((args[1] != NULL && strcmp(args[1], "-h") != 0) && (args[2] != NULL && strcmp(args[2], "-h") != 0) && args[3] == NULL) {
676
677
678
         if (copy_source_to_dest(args[1], args[2]) != 0) {
679
          u_perror("cp");
680
           return NULL;
         }
681
         return NULL;
682
683
      }
684
      P_ERRNO = P_EUNKNOWN;
u_perror("cp");
685
686
687
      return NULL;
688 }
```

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.

This function is a wrapper for k_ls, which is a kernel-level function.

Definition at line 472 of file fat_routines.c.

```
473
      // Note: we already check if fs is mounted in k\_ls
474
475
      char** args = (char**)arg;
      if (args[1] != NULL) {
476
       if (k_ls(args[1]) == -1) {
478
        u_perror("ls");
479
         return NULL;
480
     } else {
  if (k_ls(NULL) == -1) {
481
482
483
        u_perror("ls");
484
         return NULL;
485
       }
486
     }
487
     return NULL;
488
489 }
```

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 34 of file fat_routines.c.

```
35
     // validate arguments
     if (num_blocks < 1 || num_blocks > 32) {
   P_ERRNO = P_EINVAL;
36
37
38
       return -1;
39
     if (blk_size < 0 || blk_size > 4) {
41
       P_ERRNO = P_EINVAL;
42
       return -1;
43
     }
44
     // determine the file system size
45
     int block_sizes[] = {256, 512, 1024, 2048, 4096};
47
     int actual_block_size = block_sizes[blk_size];
     int fat_size = num_blocks * actual_block_size;
int fat_entries = fat_size / 2;
48
49
     int num data blocks =
50
          (num\_blocks == 32)
51
              ? fat_entries - 2
: fat_entries - 1; // note: first entry is reserved for metadata!
54
     size_t filesystem_size = fat_size + (actual_block_size * num_data_blocks);
55
     // create the file for the filesystem
56
     int fd = open(fs_name, O_RDWR | O_CREAT | O_TRUNC, 0644);
     if (fd == -1) {
59
      P_ERRNO = P_EOPEN;
60
       return -1;
61
62
    // extend the file to the required size
if (ftruncate(fd, filesystem_size) == -1) {
63
65
       P_ERRNO = P_EFUNC;
66
       close(fd);
67
       return -1;
68
69
70
     // allocate the FAT
     uint16_t* temp_fat = (uint16_t*)calloc(fat_entries, sizeof(uint16_t));
72
     if (!temp_fat) {
73
       P_ERRNO = P_EMALLOC;
74
       close(fd);
75
       return -1;
76
78
     // 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++) {</pre>
79
8.0
81
      temp_fat[i] = FAT_FREE;
82
83
84
     \ensuremath{//} write the FAT to the file
85
86
    if (write(fd, temp_fat, fat_size) != fat_size) {
      P_ERRNO = P_EWRITE;
87
88
       free(temp fat);
89
       close(fd);
90
       return -1;
91
92
93
     // initialize the root directory + write to memory
    uint8_t* root_dir = (uint8_t*)calloc(actual_block_size, 1);
94
     if (lseek(fd, fat_size, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
97
       free(temp_fat);
98
       free (root_dir);
99
       close(fd);
100
        return -1;
101
      if (write(fd, root_dir, actual_block_size) != actual_block_size) {
```

```
P_ERRNO = P_EWRITE;
103
104
       free(temp_fat);
105
       free (root_dir);
106
       close(fd);
107
        return -1;
108
109
110
     // clean up
111
     free(temp_fat);
112
     free (root_dir);
     close(fd);
113
114
     return 0;
115 }
```

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 120 of file fat_routines.c.

```
120
       // check if a filesystem is already mounted
121
       if (is_mounted) {
122
123
        P_ERRNO = P_EBUSY;
124
         return -1;
125
126
       // open the file with fs_name + set the global fs_fd \,
127
128
       fs_fd = open(fs_name, O_RDWR);
       if (fs_fd == -1) {
P_ERRNO = P_ENOENT;
129
130
131
         return -1;
132
133
       \ensuremath{//} read the first two bytes to get size configuration
134
135
       uint16_t config;
       if (read(fs_fd, &config, sizeof(config)) != sizeof(config)) {
   P_ERRNO = P_EREAD;
136
137
138
         close(fs_fd);
         fs_fd = -1;
return -1;
139
140
141
142
143
       \label{eq:continuous} \mbox{// extract FAT region size information}
144
       num_fat_blocks = (config » 8) & 0xFF; // MSB
       int block_size_config = config & 0xFF; // LSB
int block_sizes[] = {256, 512, 1024, 2048, 4096};
block_size = block_sizes[block_size_config];
145
146
147
148
       fat_size = num_fat_blocks * block_size;
149
150
       // map the FAT region into memory
       if (lseek(fs_fd, 0, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
151
152
153
         close(fs_fd);
         fs_fd = -1;
154
155
         return -1;
156
157
158
       fat = mmap(NULL, fat_size, PROT_READ | PROT_WRITE, MAP_SHARED, fs_fd, 0);
if (fat == MAP_FAILED) {
159
         P_ERRNO = P_EMAP;
160
161
         close(fs_fd);
162
          fs\_fd = -1;
163
         return -1;
164
165
      init_fd_table(fd_table); // initialize the file descriptor table
166
167
      is_mounted = true;
168 return 0;
169 }
```

4.1.1.8 mv()

```
void* mv ( \label{eq:void* arg } \mbox{void} * \mbox{\it arg } \mbox{\it )}
```

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

Arguments array (command line arguments)

Returns

557

void pointer (unused)

Definition at line 557 of file fat_routines.c.

```
char** args = (char**)arg;
558
559
560
                            // verify that the file system is mounted
                           if (!is_mounted) {
561
                                   P_ERRNO = P_EFS_NOT_MOUNTED;
u_perror("mv");
562
563
564
                                    return NULL;
565
566
567
                            // check if we have both source and destination arguments
568
                            if (args[1] == NULL \mid | args[2] == NULL) {
569
                                  P_ERRNO = P_EINVAL;
                                    u_perror("mv");
570
571
                                    return NULL;
572
573
574
                            char* source = args[1];
575
                            char* dest = args[2];
576
                            // check if they're trying to rename to the same name
577
578
                            if (strcmp(source, dest) == 0) {
579
                                    return NULL;
580
581
582
                            // check if source file exists % \left( 1\right) =\left( 1\right) \left( 1\right) \left
583
                          dir_entry_t source_entry;
int source_offset = find_file(source, &source_entry);
584
                            if (source_offset < 0) {</pre>
585
                               u_perror("mv");
return NULL;
586
587
588
589
                          // check if the destination file already exists
590
                          dir_entry_t dest_entry;
int dest_offset = find_file(dest, &dest_entry);
591
592
593
594
                             // destination file exists
595
                            if (dest offset >= 0) {
                                   // check if the destination file is currently open by any process
for (int i = 0; i < MAX_FDS; i++) {</pre>
596
597
598
                                             if (fd_table[i].in_use && strcmp(fd_table[i].filename, dest) == 0) {
599
                                                       P_ERRNO = P_EBUSY;
                                                         u_perror("mv");
600
601
                                                         return NULL;
602
603
                                    }
604
605
                                      // if destination file exists, delete it
606
                                      if (mark_entry_as_deleted(&dest_entry, dest_offset) != 0) {
607
                                             u_perror("mv");
608
                                               return NULL;
609
```

```
610
       }
611
612
       // rename file
       strncpy(source_entry.name, dest, sizeof(source_entry.name) - 1);
source_entry.name[sizeof(source_entry.name) - 1] = '\0';
613
614
615
       // write the updated entry back to disk
616
617
       if (lseek(fs_fd, source_offset, SEEK_SET) == -1) {
       P_ERRNO = P_ELSEEK;
u_perror("mv");
618
619
         return NULL;
620
621
622
623
       if (write(fs_fd, &source_entry, sizeof(source_entry)) != sizeof(source_entry)) {
        P_ERRNO = P_EWRITE;
u_perror("mv");
624
625
626
         return NULL;
      }
627
628
629
      return NULL;
630 }
```

4.1.1.9 rm()

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

Removes files.

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

Parameters

arg | Arguments array (command line arguments)

Returns

void pointer (unused)

Definition at line 693 of file fat_routines.c.

```
693
        char** args = (char**)arg;
694
695
696
        // verify that the file system is mounted
        if (!is_mounted) {
   P_ERRNO = P_EFS_NOT_MOUNTED;
   u_perror("rm");
697
698
699
          return NULL;
700
701
702
703
        // check if we have any arguments
704
        if (args[1] == NULL) {
         P_ERRNO = P_EINVAL;
u_perror("rm");
705
706
          return NULL;
707
708
709
710
        // process each file argument
        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);
711
712
713
714
715
716
           if (entry_offset < 0) {</pre>
           // file doesn't exist
P_ERRNO = P_ENOENT;
717
718
             u_perror("rm");
719
720
             continue;
```

```
721
        }
722
723
        // check if file is currently open
724
        for (int j = 0; j < MAX_FDS; j++) {</pre>
         if (fd_table[j].in_use && strcmp(fd_table[j].filename, args[i]) == 0) {
   P_ERRNO = P_EBUSY;
725
726
727
            u_perror("rm");
728
729
730
        }
731
        // mark the directory entry as deleted
732
        if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
733
734
735
          u_perror("rm");
736
          continue;
737
738
739
        char deleted = 1; // mark as deleted
740
        if (write(fs_fd, &deleted, sizeof(deleted)) != sizeof(deleted)) {
741
        P_ERRNO = P_EWRITE;
          u_perror("rm");
742
743
         continue;
744
745
746
        // free the FAT chain for this file
747
        uint16_t block = entry.firstBlock;
        while (block != FAT_FREE && block != FAT_EOF) {
748
749
        uint16_t next_block = fat[block];
          fat[block] = FAT_FREE;
750
751
          block = next block;
752
        }
753
754
755
      return NULL;
756 }
```

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 497 of file fat_routines.c.

```
498
      char** args = (char**)arg;
499
500
      \ensuremath{//} verify that the file system is mounted
501
      if (!is_mounted) {
  P_ERRNO = P_EFS_NOT_MOUNTED;
502
        u_perror("touch");
503
504
        return NULL;
505
506
      // check if we have any arguments
507
      if (args[1] == NULL) {
508
        P_ERRNO = P_EINVAL;
509
510
        u_perror("touch");
511
        return NULL;
512
513
      // process each file argument
514
      for (int i = 1; args[i] != NULL; i++) {
515
516
        dir_entry_t entry;
517
        int entry_offset = find_file(args[i], &entry);
518
        // file exists
if (entry_offset >= 0) {
  entry.mtime = time(NULL);
519
520
521
```

```
// write the updated entry back to the directory
           if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
524
525
             u_perror("touch");
526
527
             continue;
528
529
           if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
             P_ERRNO = P_EWRITE;
u_perror("touch");
530
531
532
             continue;
533
534
        } else {
535
           // file doesn't exist, create a new empty file
536
537
           // check if the fat is full
           if (P_ERRNO == P_EFULL) {
  u_perror("touch");
538
539
             return NULL;
540
541
542
543
           // add the file entry to root directory
           if (add_file_entry(args[i], 0, 0, TYPE_REGULAR, PERM_READ_WRITE) == -1) {
    u_perror("touch");
544
545
546
             continue;
547
           }
548
       }
549
      }
550
551
      return NULL;
552 }
```

4.1.1.11 unmount()

```
int unmount ( )
```

Unmounts the current filesystem and reset variables.

Unmounts the currently mounted filesystem.

Definition at line 174 of file fat routines.c.

```
174
175
       // first check that a file system is actually mounted
176
       if (!is_mounted) {
177
       P_ERRNO = P_EFS_NOT_MOUNTED;
178
         return -1;
      }
179
180
      // unmap the FAT
if (fat != NULL) {
181
182
183
       if (munmap(fat, fat_size) == -1) {
         P_ERRNO = P_EMAP;
return -1;
184
185
186
        fat = NULL;
187
188
189
      // close fs_fd
190
      if (fs_fd != -1) {
  if (close(fs_fd) == -1) {
    P_ERRNO = P_ECLOSE;
}
191
192
193
194
           return -1;
195
196
        fs\_fd = -1;
197
198
      // reset the other globals
199
      num_fat_blocks = 0;
200
201
      block_size = 0;
202
      fat_size = 0;
      is_mounted = false;
203
204
      return 0;
205 }
```

4.1.2 Variable Documentation

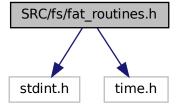
4.1.2.1 current_running_pcb

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

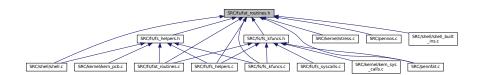
Definition at line 38 of file scheduler.c.

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 214 of file fat_routines.c.
```

```
215
              char** args = (char**)arg;
216
              \ensuremath{//} verify that the file system is mounted
217
             if (!is_mounted) {
  P_ERRNO = P_EFS_NOT_MOUNTED;
218
219
220
                  u_perror("cat");
221
                  return NULL;
222
223
             // early return if there is nothing after cat
224
             if (args[1] == NULL) {
225
226
                // if none of the above conditions, then check if we need to redirect stdin
227
                  if (current_running_pcb) {
228
                       // open new stdin
                      int in_fd = current_running_pcb->input_fd;
229
                      int out_fd = current_running_pcb->output_fd;
char* file_1 = fd_table[in_fd].filename;
230
231
                      char* file_2 = fd_table[out_fd].filename;
233
234
                       // edge case when input and output have the same file name and \ensuremath{\mathsf{we'}}\xspace representation and the same file name and <math>\ensuremath{\mathsf{we'}}\xspace representation and the same file name and the same file name and <math>\ensuremath{\mathsf{we'}}\xspace representation and the same file name file name and the same file name and the same file name file name and the same file name file nam
235
                       if ((strcmp(file_1, file_2) == 0) && is_append) {
   P_ERRNO = P_EREDIR;
236
237
                           u_perror("cat");
238
                           return NULL;
239
240
                      // edge case when input and output files names are the same but we're not appending
241
242
                       // truncates the file
                       if ((strcmp(file_1, file_2) == 0)) {
243
                           return NULL;
245
246
247
                        // get the size of stdin file
248
                       off_t in_fd_size = k_lseek(in_fd, 0, SEEK_END);
                       if (in_fd_size == -1) {
249
250
                          k_close(in_fd);
                           u_perror("cat");
252
                           return NULL;
253
254
                       if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
255
                           k close(in fd);
                           u_perror("cat");
256
                           return NULL;
258
259
                       char* buffer = (char*)malloc(block size);
260
                       if (buffer == NULL) {
261
                          P_ERRNO = P_EMALLOC;
262
263
                           k_close(in_fd);
                           u_perror("cat");
264
265
                           return NULL;
266
2.67
268
                       int bytes read;
269
                       ssize_t bytes_remaining = in_fd_size;
270
271
                       while (bytes_remaining > 0) {
                           ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
272
273
                           bytes_read = k_read(in_fd, buffer, bytes_to_read);
274
275
                           if (bytes_read <= 0) {</pre>
276
                              break;
277
278
                           if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
279
280
                               free (buffer);
281
                                k close(in fd);
                                u_perror("cat");
283
                                break;
284
```

```
285
286
             bytes_remaining -= bytes_read;
287
288
           // read error
289
           if (bytes_read < 0) {</pre>
290
            free (buffer);
291
292
              k_close(in_fd);
             u_perror("cat");
293
294
             return NULL;
           }
295
296
           k_close(in_fd);
if (out_fd != STDOUT_FILENO) {
297
298
299
             k_close(out_fd);
300
           free (buffer);
301
302
           return NULL;
303
         P_ERRNO = P_EINVAL;
u_perror("cat");
304
305
306
         return NULL;
307
308
309
       // check for output file with -w or -a flag
       int out_fd = -1;
310
311
       int out_mode = 0;
312
313
       \ensuremath{//} scan arguments and determine output fd and output mode
314
       int i;
       for (i = 1; args[i] != NULL; i++) {
315
316
         if (strcmp(args[i], "-w") == 0 && args[i + 1] != NULL) {
317
           out_mode = F_WRITE;
            out_fd = k_open(args[i + 1], F_WRITE);
318
           if (out_fd < 0) {
   u_perror("cat");</pre>
319
320
             return NULL;
321
322
323
           break;
324
         } 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>
325
326
327
            u_perror("cat");
return NULL;
328
329
330
331
           break;
332
         }
      }
333
334
335
       // if no output redirection found, use STDOUT
336
       if (out_fd < 0) {</pre>
337
        if (current_running_pcb) {
338
           out_fd = current_running_pcb->output_fd;
339
        } else {
           out_fd = STDOUT_FILENO;
340
341
342
343
      // handle small case: cat -w OUTPUT_FILE or cat -a OUTPUT_FILE (read from stdin) if ((strcmp(args[1], "-w") == 0 || strcmp(args[1], "-a") == 0) && args[2] != NULL && args[3] == NULL) {
344
345
346
347
        char buffer[1024];
348
349
        while (1) {
350
           ssize_t bytes_read = k_read(STDIN_FILENO, buffer, sizeof(buffer));
351
           if (bytes read < 0) {
352
             u_perror("cat");
if (out_fd != STDOUT_FILENO) {
353
354
355
                k_close(out_fd);
356
357
              return NULL;
358
359
360
            if (bytes_read == 0) {
361
             break;
362
363
364
           if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
             u_perror("cat");
if (out_fd != STDOUT_FILENO) {
365
366
367
               k_close(out_fd);
              }
368
369
              return NULL;
370
           }
371
         }
```

```
373
         if (out_fd != STDOUT_FILENO) {
374
           k_close(out_fd);
         }
375
376
         return NULL;
377
378
379
       // handle concatenating one or more files: cat FILE ... [-w/-a OUTPUT_FILE]
380
       int start = 1;
381
       int end = i - 1;
382
      end = i - 1; // skip the output redirection arguments
}
383
384
385
386
387
       // process each input file
       for (i = start; i <= end; i++) {
    // skip the redirection flags and their arguments
    if (strcmp(args[i], "-w") == 0 || strcmp(args[i], "-a") == 0) {</pre>
388
389
390
391
           i++;
           continue;
392
393
394
         // open the current input file
int in_fd = k_open(args[i], F_READ);
if (in_fd < 0) {</pre>
395
396
397
           u_perror("cat");
398
399
           continue;
400
401
         // 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) {
402
403
404
405
           k_close(in_fd);
406
           u_perror("cat");
407
           continue;
408
409
410
         // use lseek to reset position to 0 for reading
411
         if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
412
           k_close(in_fd);
           u_perror("cat");
413
414
           continue;
415
416
417
         // copy file content to output
         char* buffer = (char*)malloc(block_size);
if (buffer == NULL) {
418
419
           P_ERRNO = P_EMALLOC;
420
421
           k_close(in_fd);
           u_perror("cat");
422
423
           continue;
424
425
         int bytes_read;
426
         ssize_t bytes_remaining = in_fd_size;
427
428
429
         while (bytes_remaining > 0) {
430
           ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
431
           bytes_read = k_read(in_fd, buffer, bytes_to_read);
432
433
            if (bytes_read <= 0) {</pre>
434
             break;
435
436
437
            if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
438
             free (buffer);
              k close(in fd):
439
440
              u_perror("cat");
441
              break;
442
443
444
           bytes_remaining -= bytes_read;
445
446
447
          // read error
448
         if (bytes_read < 0) {</pre>
449
           free (buffer);
           k_close(in_fd);
u_perror("cat");
450
451
452
           continue;
453
454
455
         k_close(in_fd);
456
         free (buffer);
457
458
```

```
459  // close output file if not stdout
460  if (out_fd != STDOUT_FILENO) {
461    k_close(out_fd);
462  }
463
464  return NULL;
465 }
```

4.2.2.2 chmod()

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 765 of file fat_routines.c.

```
765
        char** args = (char**)arg;
if (!args || !args[0] || !args[1] || !args[2]) {
    P_ERRNO = P_EINVAL;
    return NULL;
766
767
768
769
770
771
772
        // Parse permission string
       const char* perm_str = args[1];
if (perm_str[0] != '+' && perm_str[0] != '-') {
774
775
776
777
         P_ERRNO = P_EINVAL;
return NULL;
778
       // Find the file and get its current directory entry
       dir_entry_t dir_entry;
```

```
int entry_offset = find_file(args[2], &dir_entry);
781
      if (entry_offset < 0) {
  P_ERRNO = P_ENOENT;</pre>
782
783
        return NULL;
784
785
786
787
      // Calculate new permissions
788
      uint8_t new_perm = dir_entry.perm;
      int i = 1; // Start after + or -
while (perm_str[i] != '\0') {
    switch (perm_str[i]) {
    case 'r':
789
790
791
792
793
            if (perm_str[0] == '+') {
794
              new_perm |= PERM_READ;
795
796
              new_perm &= ~PERM_READ;
797
798
             break;
           case 'w':
           if (perm_str[0] == '+') {
800
801
              new_perm |= PERM_WRITE;
           } else {
802
              new_perm &= ~PERM_WRITE;
803
804
805
             break;
806
        case 'x':
          if (perm_str[0] == '+') {
807
808
              new_perm |= PERM_EXEC;
809
           new_perm &= ~PERM_EXEC;
}
810
811
812
             break;
813
814
            P_ERRNO = P_EINVAL;
815
             return NULL;
816
817
        i++;
818
819
820
      // Update the directory entry
      dir_entry.perm = new_perm;
dir_entry.mtime = time(NULL);
821
822
823
824
      // Seek to the entry's position
      if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
return NULL;
826
827
828
829
      // Write the updated entry back
830
      if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
831
      P_ERRNO = P_EWRITE;
832
833
        return NULL;
834
835
836
      return NULL;
```

4.2.2.3 cmpctdir()

```
void* cmpctdir ( \label{eq:condition} \mbox{void} \ * \ \mbox{\it arg} \ )
```

Compacts the root directory by removing all deleted entries.

Parameters

arg | Arguments array (command line arguments)

Returns

void pointer (unused)

Compacts the root directory by removing all deleted entries.

Definition at line 846 of file fat_routines.c.

4.2.2.4 cp()

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

Copies files.

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

Usage formats:

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

Parameters

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

Returns

return 0 on success, -1 on error

Copies files.

Definition at line 635 of file fat_routines.c.

```
635
636
        char** args = (char**)arg;
637
        // check that we have enough arguments
        if (args[1] == NULL || args[2] == NULL) {
   P_ERRNO = P_EINVAL;
639
640
         u_perror("cp");
641
642
           return NULL;
643
644
       // cp -h SOURCE DEST
if (strcmp(args[1], "-h") == 0) {
  if (args[2] == NULL || args[3] == NULL) {
    P_ERRNO = P_EINVAL;
    u_perror("cp");
645
646
647
648
649
650
              return NULL;
```

```
651
         }
652
653
         if (copy_host_to_pennfat(args[2], args[3]) != 0) {
          u_perror("cp");
654
655
           return NULL;
656
657
         return NULL;
658
659
       // cp SOURCE -h DEST
660
       if (args[2] != NULL && strcmp(args[2], "-h") == 0) {
661
        if (args[3] == NULL) {
P_ERRNO = P_EINVAL;
u_perror("cp");
662
663
664
665
           return NULL;
666
667
        if (copy_pennfat_to_host(args[1], args[3]) != 0) {
  u_perror("cp");
668
669
670
           return NULL;
671
672
         return NULL;
673
      }
674
675
       // cp SOURCE DEST
676
       if ((args[1] != NULL && strcmp(args[1], "-h") != 0) && (args[2] != NULL && strcmp(args[2], "-h") != 0) && args[3] == NULL) {
677
         if (copy_source_to_dest(args[1], args[2]) != 0) {
    u_perror("cp");
678
679
           return NULL;
680
681
682
         return NULL;
683
684
      P_ERRNO = P_EUNKNOWN;
u_perror("cp");
685
686
687
      return NULL;
688 }
```

4.2.2.5 ls()

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

Lists files in the current directory.

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

Parameters

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

Returns

0 on success, -1 on error

Lists files in the current directory.

This function is a wrapper for k_ls, which is a kernel-level function.

Definition at line 472 of file fat_routines.c.

```
472 {
473  // Note: we already check if fs is mounted in k_ls
474 
475  char** args = (char**)arg;
```

```
if (args[1] != NULL) {
477
       if (k_ls(args[1]) == -1) {
478
         u_perror("ls");
         return NULL;
479
480
481
     } else {
      if (k_ls(NULL) == -1) {
482
483
         u_perror("ls");
         return NULL;
484
485
     }
486
487
488
     return NULL;
```

4.2.2.6 mkfs()

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 34 of file fat_routines.c.

```
35
     // validate arguments
     if (num_blocks < 1 || num_blocks > 32) {
   P_ERRNO = P_EINVAL;
36
       return -1;
40
     if (blk_size < 0 || blk_size > 4) {
       P_ERRNO = P_EINVAL;
return -1;
41
42
43
45
     // determine the file system size
46
     int block_sizes[] = {256, 512, 1024, 2048, 4096};
     int actual_block_size = block_sizes[blk_size];
     int fat_size = num_blocks * actual_block_size;
int fat_entries = fat_size / 2;
48
49
50
    int num_data_blocks =
          (num\_blocks == 32)
             ? fat_entries - 2
: fat_entries - 1; // note: first entry is reserved for metadata!
53
     size_t filesystem_size = fat_size + (actual_block_size * num_data_blocks);
54
55
     // create the file for the filesystem
     int fd = open(fs_name, O_RDWR | O_CREAT | O_TRUNC, 0644);
     if (fd == -1) {
    P_ERRNO = P_EOPEN;
59
60
       return -1;
61
62
     // extend the file to the required size
```

```
if (ftruncate(fd, filesystem_size) == -1) {
       P_ERRNO = P_EFUNC;
66
        close(fd);
67
        return -1;
68
69
     // allocate the FAT
70
71
      uint16_t* temp_fat = (uint16_t*)calloc(fat_entries, sizeof(uint16_t));
     if (!temp_fat) {
  P_ERRNO = P_EMALLOC;
72
73
       close(fd);
74
75
        return -1;
76
     }
78
      \ensuremath{//} initialize FAT entries to their correct values
     temp_fat[0] = (num_blocks « 8) | blk_size;
temp_fat[1] = FAT_EOF;
for (int i = 2; i < fat_entries; i++) {
  temp_fat[i] = FAT_FREE;
79
80
81
82
84
85
     // write the FAT to the file
     if (write(fd, temp_fat, fat_size) != fat_size) {
   P_ERRNO = P_EWRITE;
86
87
88
        free(temp_fat);
89
        close(fd);
90
        return -1;
     }
91
92
     // initialize the root directory + write to memory
93
     uint8_t* root_dir = (uint8_t*)calloc(actual_block_size, 1);
if (lseek(fd, fat_size, SEEK_SET) == -1) {
    P_ERRNO = P_ELSEEK;
94
97
        free(temp_fat);
98
        free(root_dir);
99
       close(fd);
100
         return -1;
101
102
       if (write(fd, root_dir, actual_block_size) != actual_block_size) {
        P_ERRNO = P_EWRITE;
103
104
         free(temp_fat);
105
         free(root_dir);
106
        close(fd);
107
         return -1;
108
109
110
      // clean up
111
      free(temp_fat);
      free (root_dir);
112
113
      close(fd);
114
       return 0;
115 }
```

4.2.2.7 mount()

```
int mount ( {\tt const\ char\ *\ fs\_name\ )}
```

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

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

Parameters

fs_name The name of the filesystem file to mount.

Returns

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

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

Definition at line 120 of file fat_routines.c.

```
// check if a filesystem is already mounted
121
       if (is_mounted) {
  P_ERRNO = P_EBUSY;
122
123
124
         return -1;
125
126
127
       // open the file with fs_name + set the global fs_fd
       fs_fd = open(fs_name, O_RDWR);
if (fs_fd == -1) {
128
129
       P_ERRNO = P_ENOENT;
130
131
        return -1;
132
133
      \ensuremath{//} read the first two bytes to get size configuration
134
      uint16_t config;
if (read(fs_fd, &config, sizeof(config)) != sizeof(config)) {
135
136
137
        P_ERRNO = P_EREAD;
138
         close(fs_fd);
139
         fs\_fd = -1;
        return -1;
140
141
142
       // extract FAT region size information
144
       num_fat_blocks = (config » 8) & 0xFF; // MSB
145
       int block_size_config = config & 0xFF; // LSB
      int block_sizes[] = {256, 512, 1024, 2048, 4096};
block_size = block_sizes[block_size_config];
fat_size = num_fat_blocks * block_size;
146
147
148
149
150
       // map the FAT region into memory
       if (lseek(fs_fd, 0, SEEK_SET) == -1) {
151
152
       P_ERRNO = P_ELSEEK;
153
        close(fs_fd);
fs_fd = -1;
154
155
         return -1;
156
157
158
       fat = mmap(NULL, fat_size, PROT_READ | PROT_WRITE, MAP_SHARED, fs_fd, 0);
      if (fat == MAP_FAILED) {
  P_ERRNO = P_EMAP;
159
160
161
        close(fs fd);
162
        fs_fd = -1;
163
        return -1;
164
165
      init_fd_table(fd_table); // initialize the file descriptor table
166
167
     is_mounted = true;
168
      return 0;
169 }
```

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 557 of file fat routines.c.
557
      char** args = (char**)arg;
558
559
560
      // verify that the file system is mounted
561
      if (!is_mounted) {
       P_ERRNO = P_EFS_NOT_MOUNTED;
u_perror("mv");
562
563
        return NULL;
564
565
566
567
       // check if we have both source and destination arguments
568
      if (args[1] == NULL \mid | args[2] == NULL) {
       P_ERRNO = P_EINVAL;
u_perror("mv");
569
570
571
        return NULL;
572
573
574
      char* source = args[1];
575
      char* dest = args[2];
576
577
      // check if they're trying to rename to the same name
578
      if (strcmp(source, dest) == 0) {
579
        return NULL;
580
581
      // check if source file exists
582
      dir_entry_t source_entry;
int source_offset = find_file(source, &source_entry);
583
584
585
      if (source_offset < 0) {</pre>
586
      u_perror("mv");
587
        return NULL;
588
589
590
      // check if the destination file already exists
      dir_entry_t dest_entry;
592
      int dest_offset = find_file(dest, &dest_entry);
593
      // destination file exists
594
      if (dest_offset >= 0) {
595
596
        // check if the destination file is currently open by any process
597
        for (int i = 0; i < MAX_FDS; i++) {</pre>
598
          if (fd_table[i].in_use && strcmp(fd_table[i].filename, dest) == 0) {
            P_ERRNO = P_EBUSY;
u_perror("mv");
599
600
601
             return NULL;
602
          }
603
604
605
         // if destination file exists, delete it
        if (mark_entry_as_deleted(&dest_entry, dest_offset) != 0) {
  u_perror("mv");
606
607
608
          return NULL;
609
        }
610
611
612
      // rename file
613
      strncpy(source_entry.name, dest, sizeof(source_entry.name) - 1);
614
      source_entry.name[sizeof(source_entry.name) - 1] = '\0';
615
616
      // write the updated entry back to disk
617
      if (lseek(fs_fd, source_offset, SEEK_SET) == -1) {
618
        P_ERRNO = P_ELSEEK;
        u_perror("mv");
619
620
        return NULL;
621
622
623
      if (write(fs_fd, &source_entry, sizeof(source_entry)) != sizeof(source_entry)) {
      P_ERRNO = P_EWRITE;
u_perror("mv");
624
625
        return NULL;
626
627
628
629
      return NULL;
630 }
```

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 693 of file fat routines.c.

```
char** args = (char**)arg;
694
695
       \ensuremath{//} verify that the file system is mounted
696
697
       if (!is_mounted) {
        P_ERRNO = P_EFS_NOT_MOUNTED;
698
699
         u_perror("rm");
700
         return NULL;
701
702
      // check if we have any arguments
703
       if (args[1] == NULL) {
704
705
        P_ERRNO = P_EINVAL;
706
         u_perror("rm");
707
         return NULL;
708
709
710
       // process each file argument
       for (int i = 1; args[i] != NULL; i++) {
   // find the file in the directory
711
712
713
         dir_entry_t entry;
714
         int entry_offset = find_file(args[i], &entry);
715
716
         if (entry_offset < 0) {</pre>
          // file doesn't exist
P_ERRNO = P_ENOENT;
717
718
719
          u_perror("rm");
720
           continue;
721
722
723
         // check if file is currently open
724
         for (int j = 0; j < MAX_FDS; j++) {</pre>
725
               (fd_table[j].in_use && strcmp(fd_table[j].filename, args[i]) == 0) {
             P_ERRNO = P_EBUSY;
u_perror("rm");
726
727
728
              continue:
729
           }
730
         }
731
732
         // mark the directory entry as deleted
         if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
733
734
           u_perror("rm");
735
736
           continue;
737
738
         char deleted = 1; // mark as deleted
if (write(fs_fd, &deleted, sizeof(deleted)) != sizeof(deleted)) {
   P_ERRNO = P_EWRITE;
739
740
741
742
           u_perror("rm");
743
           continue;
744
745
         // free the FAT chain for this file
uint16_t block = entry.firstBlock;
while (block != FAT_FREE && block != FAT_EOF) {
746
747
748
749
           uint16_t next_block = fat[block];
```

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 497 of file fat_routines.c.

```
498
       char** args = (char**)arg;
499
500
       // verify that the file system is mounted
      if (!is_mounted) {
   P_ERRNO = P_EFS_NOT_MOUNTED;
   u_perror("touch");
501
502
503
504
         return NULL;
505
506
507
      // check if we have any arguments
508
      if (args[1] == NULL) {
        P_ERRNO = P_EINVAL;
u_perror("touch");
509
510
511
         return NULL;
512
513
       // process each file argument
514
       for (int i = 1; args[i] != NULL; i++) {
    dir_entry_t entry;
515
516
517
        int entry_offset = find_file(args[i], &entry);
518
        // file exists
if (entry_offset >= 0) {
519
520
521
           entry.mtime = time(NULL);
522
523
           // write the updated entry back to the directory
           if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
524
525
             u_perror("touch");
526
527
             continue;
528
            if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
   P_ERRNO = P_EWRITE;
529
530
              u_perror("touch");
531
532
              continue;
```

```
534
          } else {
535
             // file doesn't exist, create a new empty file
536
             // check if the fat is full
if (P_ERRNO == P_EFULL) {
  u_perror("touch");
  return NULL;
537
538
539
541
542
             // add the file entry to root directory
if (add_file_entry(args[i], 0, 0, TYPE_REGULAR, PERM_READ_WRITE) == -1) {
  u_perror("touch");
543
544
545
546
                continue;
547
548
549 }
550
       return NULL;
551
```

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 174 of file fat routines.c.
```

```
174
      // first check that a file system is actually mounted
176
      if (!is_mounted) {
       P_ERRNO = P_EFS_NOT_MOUNTED;
177
178
        return -1;
179
180
      // unmap the FAT
if (fat != NULL) {
181
      if (munmap(fat, fat_size) == -1) {
183
184
        P_ERRNO = P_EMAP;
        return -1;
185
186
187
        fat = NULL;
188
189
190
      // close fs_fd
      if (fs_fd != -1) {
  if (close(fs_fd) == -1) {
    P_ERRNO = P_ECLOSE;
}
191
192
193
194
          return -1;
195
196
        fs\_fd = -1;
197
198
199
      // reset the other globals
200
      num_fat_blocks = 0;
201
      block_size = 0;
202
      fat_size = 0;
203
      is_mounted = false;
2.04
     return 0;
205 }
```

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 <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;
272
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;
279
        return -1;
280
281
282
283
       // start with root directory block (block 1)
284
      uint16_t current_block = 1;
      int offset = 0;
dir_entry_t dir_entry;
285
286
287
288
       while (1) {
       // position at the start of current block of the root directory
if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
289
290
         P_ERRNO = P_ELSEEK;
return -1;
291
292
293
294
295
         // reset offset for new block
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
301
            P_ERRNO = P_EREAD;
302
             return -1;
303
           }
304
305
           // found a free slot
306
           if (dir_entry.name[0] == 0 || dir_entry.name[0] == 1) {
             // initialize the new entry
```

```
memset(&dir_entry, 0, sizeof(dir_entry));
             strncpy(dir_entry.name, filename, 31);
dir_entry.size = size;
309
310
             dir_entry.firstBlock = first_block;
311
            dir_entry.type = type;
dir_entry.perm = perm;
dir_entry.mtime = time(NULL);
312
313
314
315
             // write the entry
316
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
332
        // current block is full, check if there's a next block
333
        if (fat[current_block] != FAT_EOF)
334
          current_block = fat[current_block];
335
          continue;
336
337
338
         // allocate a new block for the root directory
        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);
351
        if (!zero_block) {
352
          P_ERRNO = P_EINVAL;
353
           return -1;
354
355
        // write this new block to the file system
356
357
        if (lseek(fs_fd, fat_size + (new_block - 1) * block_size, SEEK_SET) == -1) {
          P_ERRNO = P_ELSEEK;
358
359
           free(zero_block);
360
           return -1;
361
        if (write(fs_fd, zero_block, block_size) != block_size) {
   P_ERRNO = P_EWRITE;
362
363
           free(zero_block);
364
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.type = type,
dir_entry.perm = perm;
dir_entry.mtime = time(NULL);
376
377
378
379
        \ensuremath{//} 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;
           return -1;
386
        }
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 i;</pre>
168
169
170
171
172
              }
173
          }
174
          if (compact_directory() == 0) {
  for (int i = 2; i < fat_size / 2; i++) {
    if (fat[i] == FAT_FREE) {
      fat[i] = FAT_EOF;
    }
}</pre>
175
176
177
178
179
                      return i;
180
            }
181
182 }
183
184
          return 0;
185 }
```

4.3.1.3 compact_directory()

```
int compact_directory ( )
```

Compacts a directory.

Compacts the root directory by removing all deleted entries.

Definition at line 692 of file fs_helpers.c.

```
692
693
      if (!is mounted) {
       P_ERRNO = P_EFS_NOT_MOUNTED;
694
        return -1;
695
696
697
698
      // buffer for temp storage of a block
699
      uint8_t* dir_buffer = malloc(block_size);
700
      if (!dir_buffer) {
  P_ERRNO = P_EMALLOC;
701
        return -1;
702
703
704
705
      // start at root directory
      uint16_t current_block = 1;
int dir_entries_count = 0;
706
707
708
      int deleted_entries_count = 0;
709
710
      \ensuremath{//} calculate number of entries and deleted entries in the root directory
711
712
      while (current_block != FAT_EOF) {
       if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
        P_ERRNO = P_ELSEEK;
free(dir_buffer);
713
714
715
          return -1;
716
717
        if (read(fs_fd, dir_buffer, block_size) != block_size) {
   P_ERRNO = P_EREAD;
718
719
720
          free(dir_buffer);
721
          return -1;
```

```
724
        // count entries and deleted entries in this block
725
        for (int offset = 0; offset < block_size; offset += sizeof(dir_entry_t)) {</pre>
          dir_entry_t* entry = (dir_entry_t*) (dir_buffer + offset);
726
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
735
          // check if it's a deleted entry
736
          if (entry->name[0] == 1) {
737
            deleted_entries_count++;
738
739
740
741
        // move onto next block, if there is one
742
        if (fat[current_block] != FAT_EOF)
743
         current_block = fat[current_block];
744
745
         break;
746
747
748
749
      // if no deleted entries, no compaction needed
750
      if (deleted_entries_count == 0) {
7.5.1
       free(dir_buffer);
752
       return 0;
753
754
755
      // allocate space for all valid entries
756
      dir_entry_t* all_entries = malloc(dir_entries_count * sizeof(dir_entry_t));
      if (!all_entries) {
  P_ERRNO = P_EMALLOC;
757
758
       free (dir_buffer);
759
760
        return -1;
761
762
763
      \ensuremath{//} 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) {
768
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
769
         P_ERRNO = P_ELSEEK;
770
          free(dir_buffer);
771
          free(all_entries);
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
        \ensuremath{//} process entries in this block
        for (int offset = 0; offset < block_size; offset += sizeof(dir_entry_t)) {
783
          dir_entry_t* entry = (dir_entry_t*) (dir_buffer + offset);
784
785
786
          // check if we've reached the end of directory
787
          if (entry->name[0] == 0) {
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
          memcpy(&all_entries[valid_entry_idx++], entry, sizeof(dir_entry_t));
797
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
      // rewrite the directory with only valid entries
808
809
      current block = 1:
```

```
int entries_per_block = block_size / sizeof(dir_entry_t);
      int blocks_needed = (valid_entry_idx + entries_per_block - 1) / entries_per_block;
811
812
813
      // clean up any excess directory blocks in the FAT chain
      uint16_t next_block = fat[current_block];
if (blocks_needed == 1) {
814
815
        // only need one block, free all others
816
817
        while (next_block != FAT_EOF) {
818
          uint16_t temp = fat[next_block];
          fat[next_block] = FAT_FREE;
819
820
          next_block = temp;
821
        fat[current_block] = FAT_EOF;
822
823
      } else {
824
        // navigate through needed blocks
825
        int block_count = 1;
        uint16_t prev_block = current_block;
826
827
828
        while (block_count < blocks_needed) {</pre>
829
         if (next_block == FAT_EOF)
830
             // need to allocate a new block
831
            uint16_t new_block = allocate_block();
            if (new_block == 0) {
  P ERRNO = P EFULL;
832
833
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
842
          prev_block = next_block;
          next_block = fat[next_block];
843
844
         block_count++;
845
846
847
        // free any excess blocks
848
        fat[prev_block] = FAT_EOF;
849
        while (next_block != FAT_EOF) {
850
          uint16_t temp = fat[next_block];
          fat[next_block] = FAT_FREE;
8.5.1
852
          next block = temp;
853
854
855
856
      // write the valid entries back to the directory blocks
857
      current block = 1;
      int entries_written = 0;
858
859
860
      while (entries_written < valid_entry_idx) {</pre>
861
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
862
          P ERRNO = P ELSEEK;
863
          free(dir_buffer);
864
          free (all_entries);
865
          return -1;
866
867
868
        memset(dir_buffer, 0, block_size);
869
870
        // copy entries to the buffer
871
        int entries_in_this_block = 0;
        while (entries_written < valid_entry_idx && entries_in_this_block < entries_per_block) {
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++;
         entries_in_this_block++;
877
878
880
        \ensuremath{//} write the buffer to the file system
881
        if (write(fs_fd, dir_buffer, block_size) != block_size) {
882
          P_ERRNO = P_EINVAL;
          free(dir_buffer);
883
884
          free(all entries);
          return -1;
885
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);
895
      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) {
437
         P_ERRNO = P_EFS_NOT_MOUNTED;
         return -1;
438
439
440
441
       // open the host file
       int host_fd = open(host_filename, O_RDONLY);
if (host_fd == -1) {
   P_ERRNO = P_EOPEN;
442
443
444
445
        return -1;
446
447
448
       \ensuremath{//} determine file size by seeking to the end and getting position
       off_t host_file_size_in_bytes = lseek(host_fd, 0, SEEK_END);
if (host_file_size_in_bytes == -1) {
   P_ERRNO = P_ELSEEK;
449
450
451
         close(host_fd);
452
453
         return -1;
454
455
      // go back to beginning of file for reading
if (lseek(host_fd, 0, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
456
457
458
         close(host_fd);
459
         return -1;
460
461
462
      // open the destination file in PennFAT int pennfat_fd = k_open (pennfat_filename, F_WRITE); if (pennfat_fd < 0) {
463
464
465
466
        close (host_fd);
467
         return -1;
468
469
       // copy the data into this buffer
uint8_t* buffer = (uint8_t*)malloc(block_size);
470
471
472
       if (!buffer) {
473
         P_ERRNO = P_EMALLOC;
474
         k_close(pennfat_fd);
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
       // read from host file
482
483
       while (bytes_remaining > 0) {
484
         // ensure bytes to read never exceeds the block size
485
         ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
486
         bytes_read = read(host_fd, buffer, bytes_to_read);
487
488
         if (bytes_read <= 0) {</pre>
489
           break;
490
491
492
         // write to pennfat_fd using k_write
493
         if (k_write(pennfat_fd, (const char*)buffer, bytes_read) != bytes_read) {
494
           free (buffer);
495
           k_close(pennfat_fd);
496
           close(host_fd);
497
           return -1;
```

```
498
       }
499
500
       bytes_remaining -= bytes_read;
501
502
     // check for read error
503
     if (bytes_read < 0) {</pre>
504
505
       P_ERRNO = P_EREAD;
506
       free(buffer);
507
       k_close(pennfat_fd);
      close(host_fd);
508
509
       return -1;
510
511
512
     // otherwise, cleanup and return success
513
     free (buffer);
514
     k_close(pennfat_fd);
     close(host_fd);
515
516
     return 0;
```

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.

```
524
      if (!is_mounted) {
525
        P_ERRNO = P_EFS_NOT_MOUNTED;
        return -1;
526
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
      // 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
542
      // go back to beginning of file for reading
543
      if (k_lseek(pennfat_fd, 0, SEEK_SET) == -1) {
544
       k_close(pennfat_fd);
545
        return -1;
546
547
548
      // open the host file
549
      int host_fd = open(host_filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
      if (host_fd == -1) {
P_ERRNO = P_EOPEN;
550
551
552
        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) {
  P_ERRNO = P_EMALLOC;
558
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;
     ssize_t bytes_read;
```

```
567
      // read from PennFAT file and write to host file
569
      while (bytes_remaining > 0) {
570
       // ensure bytes to read never exceeds the block size
        ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;
bytes_read = k_read(pennfat_fd, buffer, bytes_to_read);
571
572
573
574
        if (bytes_read <= 0) {</pre>
575
          break;
576
577
        if (write(host_fd, buffer, bytes_read) != bytes_read) {
578
579
          P_ERRNO = P_EINVAL;
580
          free (buffer);
581
           close(host_fd);
582
          k_close(pennfat_fd);
583
          return -1;
584
        }
585
586
        bytes_remaining -= bytes_read;
587
588
      // check for read error
589
      if (bytes_read < 0) {
  P_ERRNO = P_EREAD;</pre>
590
591
592
        free (buffer);
        close(host_fd);
593
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);
602
      return 0;
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.

```
609
      if (!is_mounted) {
        P_ERRNO = P_EFS_NOT_MOUNTED;
612
        return -1;
613
614
615
     // open the source file
     int source_fd = k_open(source_filename, F_READ);
if (source_fd < 0) {</pre>
616
617
618
       return -1;
619
620
621
      // get the source file size
      off_t source_file_size_in_bytes = k_lseek(source_fd, 0, SEEK_END);
622
623
      if (source_file_size_in_bytes == -1) {
624
      k_close(source_fd);
625
       return -1;
     }
626
627
628
      // 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
     // open the destination file
     int dest_fd = k_open(dest_filename, F_WRITE);
```

```
636
      if (dest_fd < 0) {</pre>
      k_close(source_fd);
637
638
        return -1;
639
640
641
      // read from source to destination
      char* buffer = (char*)malloc(block_size);
642
643
      if (!buffer) {
644
       P_ERRNO = P_EMALLOC;
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;
652
653
      while (bytes_remaining > 0) {
654
       // make sure the bytes to read doesn't exceed block size
655
        ssize_t bytes_to_read = bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
656
        bytes_read = k_read(source_fd, buffer, bytes_to_read);
657
658
       ωyte
break;
}
       if (bytes_read <= 0) {</pre>
659
660
661
662
        if (k_write(dest_fd, buffer, bytes_read) != bytes_read) {
        free (buffer);
663
664
          k_close(source_fd);
665
         k_close(dest_fd);
666
          return -1:
667
        }
668
669
      // check for read error
if (bytes_read < 0) {</pre>
670
671
672
       free (buffer);
673
        k_close(source_fd);
674
        k_close(dest_fd);
675
       return -1;
676
677
     // otherwise, cleanup and return success
678
     free (buffer);
      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
      P_ERRNO = P_EBADF;
109
110
        return -1;
111
112
113
      if (!fd_table[fd].in_use) {
      P_ERRNO = P_EBADF;
114
        return -1;
115
116
117
      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;
        fd_table[fd].first_block = 0;
```

```
124     fd_table[fd].position = 0;
125     fd_table[fd].mode = 0;
126    }
127    return fd_table[fd].ref_count;
128 }
```

4.3.1.8 find file()

Searches for a file in the root directory.

Retrieves the file's absolute offset in the filesystem.

Definition at line 192 of file fs helpers.c.

```
{
192
193
      if (!is_mounted) {
194
       P_ERRNO = P_EFS_NOT_MOUNTED;
195
        return -1;
196
197
      // Start with root directory block (block 1)
198
199
      uint16_t current_block = 1;
      int offset_in_block = 0;
201
      int absolute_offset = 0;
202
      dir_entry_t dir_entry;
203
204
      while (1) {
205
       // Position at the start of current block
206
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
207
         P_ERRNO = P_ELSEEK;
208
          return -1;
209
210
211
        // reset offset for new block
212
        offset_in_block = 0;
213
214
        // calculate the absolute offset
215
        absolute_offset = fat_size + (current_block - 1) * block_size;
216
217
        // search current block
        while (offset_in_block < block_size) {</pre>
218
219
         if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
220
           P_ERRNO = P_EREAD;
221
            return -1;
222
          }
223
224
          // check if we've reached the end of directory
225
          if (dir_entry.name[0] == 0) {
226
            break;
          }
227
228
          // check if this is a deleted entry
229
          if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
  offset_in_block += sizeof(dir_entry);
  absolute_offset += sizeof(dir_entry);
230
231
232
233
            continue;
          }
234
235
236
          // check if we found the file
          if (strcmp(dir_entry.name, filename) == 0) {
237
238
            if (entry) {
239
              memcpy(entry, &dir_entry, sizeof(dir_entry));
240
241
            return absolute_offset; // return the absolute file offset
242
243
244
          offset_in_block += sizeof(dir_entry);
245
          absolute_offset += sizeof(dir_entry);
246
2.47
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];
```

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.

4.3.1.10 has_executable_permission()

```
int has_executable_permission ( \quad \text{int } fd \ )
```

Checks if a file has executable permissions.

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

Definition at line 133 of file fs_helpers.c.

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

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;
      fd_table[0].ref_count = 1;
46
      strncpy(fd_table[0].filename, "<stdin>", 31);
fd_table[0].mode = F_READ;
47
48
49
      // STDOUT (fd 1)
      fd_table[1].in_use = 1;
      strncpy(fd_table[1].filename, "<stdout>", 31);
fd_table[1].mode = F_WRITE; // write-only
fd_table[1].ref_count = 1;
52
53
54
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
58
59
60
     fd_table[2].ref_count = 1;
62
     // other file descriptors (fd 3 and above)
      for (int i = 3; i < MAX_FDS; i++) {</pre>
       fd_table[i].in_use = 0;
64
        fd_table[i].ref_count = 0;
memset(fd_table[i].filename, 0, sizeof(fd_table[i].filename));
fd_table[i].size = 0;
6.5
66
67
         fd_table[i].first_block = 0;
         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.

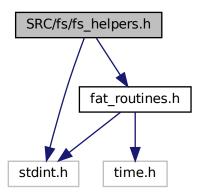
4.3.2.8 num_fat_blocks

```
int num_fat_blocks = 0
```

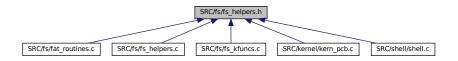
Definition at line 29 of file fs_helpers.c.

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;
     // 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

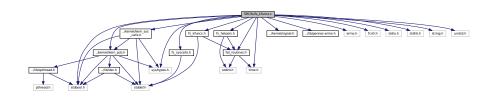
```
int num_fat_blocks [extern]
```

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.

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 k_close()

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

Kernel-level call to close a file.

Closes an open file.

Definition at line 528 of file fs_kfuncs.c.

```
528
529
       // validate the file descriptor
      if (fd < 0 || fd >= MAX_FDS) {
   P_ERRNO = P_EBADF;
   return -1;
530
531
532
533
534
      // ensure any pending changes are written to disk
535
      // update the directory entry with the current file size
536
      dir_entry_t entry;
int file_offset = find_file(fd_table[fd].filename, &entry);
537
539
       if (file_offset >= 0) {
540
       // update file size if it changed
if (entry.size != fd_table[fd].size) {
  entry.size = fd_table[fd].size;
541
542
543
544
           entry.mtime = time(NULL);
545
546
           if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
547
             P_ERRNO = P_ELSEEK;
              return -1;
548
549
            if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
```

```
P_ERRNO = P_EWRITE;
551
           return -1;
552
553
         }
       }
554
555
     }
556
557
      // decrement the reference count
558
     decrement_fd_ref_count(fd);
559
     return 0;
560
561 }
```

4.5.1.2 k_ls()

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

Kernel-level call to list files.

Lists files or file information.

Definition at line 668 of file fs_kfuncs.c.

```
668
669
      if (!is mounted) {
670
       P_ERRNO = P_EFS_NOT_MOUNTED;
671
        return -1;
672
673
      // start with root directory block
uint16_t current_block = 1;
dir_entry_t dir_entry;
674
675
676
677
      uint32_t offset = 0;
678
679
      // if filename is null, list all files in the current directory
680
      if (filename == NULL) {
681
        while (1) {
          // adjust pointer to beginning of current block
682
           if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
683
684
            P_ERRNO = P_ELSEEK;
685
             return -1;
686
          }
687
688
          offset = 0;
689
690
           // search current block
691
           while (offset < block_size) {</pre>
692
             if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
693
               P_ERRNO = P_EREAD;
694
               return -1;
695
696
697
             // check if we've reached the end of directory
698
             if (dir_entry.name[0] == 0) {
699
               break;
700
701
702
             // skip deleted entries
703
             if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
704
               offset += sizeof(dir_entry);
705
               continue;
706
707
708
             // format permission string
709
             char perm_str[4] = "---";
             if (dir_entry.perm & PERM_READ)
  perm_str[0] = 'r';
710
711
            if (dir_entry.perm & PERM_WRITE)
perm_str[1] = 'w';
if (dir_entry.perm & PERM_EXEC)
perm_str[2] = 'x';
712
713
714
715
716
717
             // format time
718
             struct tm* tm_info = localtime(&dir_entry.mtime);
719
             char time str[50];
720
             strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);
721
```

```
722
             // print entry details
723
             char buffer[128];
724
             int len;
725
            if (dir_entry.firstBlock == 0) {
              len = snprintf(buffer, sizeof(buffer), " -%s- %6d %s %s\n",
72.6
                       perm_str, dir_entry.size, time_str, dir_entry.name);
727
728
729
               len = snprintf(buffer, sizeof(buffer), "%2d -%s- %6d %s %s\n",
730
                       dir_entry.firstBlock, perm_str, dir_entry.size, time_str, dir_entry.name);
731
732
            if (len < 0 || len >= (int)sizeof(buffer)) {
733
              P_ERRNO = P_EUNKNOWN;
734
735
              return -1;
736
737
            if (k_write(STDOUT_FILENO, buffer, len) != len) {
738
739
              P ERRNO = P EWRITE;
              return -1;
740
741
742
743
            offset += sizeof(dir_entry);
744
          }
745
746
          // move to the next block if there is one
          if (fat[current_block] != FAT_EOF)
747
748
            current_block = fat[current_block];
749
            continue;
750
751
752
          // no more blocks to search
753
          break;
754
755
      } else {
        // find and display specific file
int file_offset = find_file(filename, &dir_entry);
if (file_offset < 0) {</pre>
756
757
758
          P_ERRNO = P_ENOENT;
759
760
          return -1;
761
762
763
        if (dir_entry.name[0] == 0) {
          P_ERRNO = P_ENOENT;
764
          return -1;
765
766
767
768
        // skip deleted entries
        if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
769
770
          P_ERRNO = P_ENOENT;
          return -1;
771
772
773
774
        // format permission string
775
        char perm_str[4] = "---";
        if (dir_entry.perm & PERM_READ)
   perm_str[0] = 'r';
776
777
778
        if (dir_entry.perm & PERM_WRITE)
779
          perm_str[1] = 'w';
        if (dir_entry.perm & PERM_EXEC)
  perm_str[2] = 'x';
780
781
782
783
        // format time
784
        struct tm* tm_info = localtime(&dir_entry.mtime);
785
        char time_str[50];
786
        strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);
787
788
        // print entry details
        char buffer[128];
789
        int len;
790
791
           (dir_entry.firstBlock == 0) {
          len = snprintf(buffer, sizeof(buffer), " -%s- %6d %s %s\n",
792
793
                   perm_str, dir_entry.size, time_str, dir_entry.name);
794
        } else {
          len = snprintf(buffer, sizeof(buffer), "%2d -%s- %6d %s %s\n",
795
                   dir_entry.firstBlock, perm_str, dir_entry.size, time_str, dir_entry.name);
796
797
798
799
        if (len < 0 || len >= (int)sizeof(buffer)) {
          P_ERRNO = P_EUNKNOWN;
800
          return -1:
801
802
803
804
        if (k_write(STDOUT_FILENO, buffer, len) != len) {
805
          P_ERRNO = P_EWRITE;
806
          return -1;
807
808
      }
```

```
809
810 return 0;
811 }
```

4.5.1.3 k_lseek()

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

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

Repositions the file offset of an open file.

```
Definition at line 622 of file fs_kfuncs.c.
```

```
622
623
       // standard file descriptors don't support lseek
if (fd == STDIN_FILENO || fd == STDOUT_FILENO || fd == STDERR_FILENO) {
624
        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) {
   P_ERRNO = P_EBADF;
629
630
631
632
         return -1;
633
634
635
       // calculate new position based on whence
636
      int32_t new_position;
637
638
       switch (whence) {
639
        case SEEK_SET:
          new_position = offset;
640
         break;
case SEEK_CUR:
641
642
          new_position = fd_table[fd].position + offset;
643
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
      // check if new position is valid
if (new_position < 0) {
  P_ERRNO = P_EINVAL;</pre>
653
654
655
656
         return -1;
657
658
      // update file position
fd_table[fd].position = new_position;
659
660
661
662
      return new_position;
663 }
```

4.5.1.4 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
       return -1;
38
39
     if ((mode & (F_READ | F_WRITE | F_APPEND)) == 0) {
40
       P_ERRNO = P_EINVAL;
       return -1;
41
42
43
     // check if the file system is mounted
45
     if (!is_mounted) {
46
     P_ERRNO = P_EFS_NOT_MOUNTED;
47
       return -1;
     }
48
49
50
     // get a free file descriptor
     int fd = get_free_fd(fd_table);
     if (fd < 0) {
52
       P_ERRNO = P_EFULL; // no free file descriptors
53
       return -1;
54
     }
55
57
     // check if the file exists
    dir_entry_t entry;
int file_offset = find_file(fname, &entry);
58
59
60
     // file exists
61
     if (file_offset >= 0) {
       // check if the file is already open in write mode by another descriptor
64
       if ((mode & (F_WRITE | F_APPEND)) != 0) {
65
          for (int i = 0; i < MAX_FDS; i++) {</pre>
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) {
67
68
              P_ERRNO = P_EBUSY; // file is already open for writing
70
              return -1;
71
           }
         }
72
73
74
75
        // fill in the file descriptor entry
76
        fd_table[fd].in_use = 1;
77
       fd_table[fd].ref_count++;
78
       strncpy(fd_table[fd].filename, fname, 31);
        fd_table[fd].filename[31] = ' \setminus 0';
79
       fd_table[fd].size = entry.size;
80
        fd_table[fd].first_block = entry.firstBlock;
       fd_table[fd].mode = mode;
83
84
        // set the initial position
8.5
       if (mode & F APPEND) {
         fd_table[fd].position = entry.size;
86
       } else {
         fd_table[fd].position = 0;
89
90
       // if mode includes {\tt F\_WRITE} and not {\tt F\_APPEND} , truncate the file
91
       if ((mode & F_WRITE) && !(mode & F_APPEND)) {
   // free all blocks except the first one
92
93
          uint16_t block = entry.firstBlock;
95
          uint16_t next_block;
96
          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
             // free the rest of the chain
103
             while (block != 0 && block != FAT_EOF) {
              next_block = fat[block];
104
               fat[block] = FAT_FREE;
105
106
               block = next_block;
107
             }
108
109
           // update file size to 0
110
           fd_table[fd].size = 0;
111
           entry.size = 0;
112
113
           entry.mtime = time(NULL);
114
115
           // update the file system with the truncated file
```

```
if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
116
117
            P_ERRNO = P_ELSEEK;
118
             return -1;
119
           if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
   P_ERRNO = P_EWRITE;
120
121
             return -1;
122
123
124
125
      } else {
         // file doesn't exist
126
127
         // we can only create it if we are reading the file if (!(mode & F_{write})) {
128
129
130
          P_ERRNO = P_ENOENT;
131
           return -1;
132
133
134
         // allocate the first block
135
         uint16_t first_block = allocate_block();
136
         if (first_block == 0) {
137
          P_ERRNO = P_EFULL;
          return -1;
138
139
140
         // create a new file entry
141
142
         if (add_file_entry(fname, 0, first_block, TYPE_REGULAR, PERM_READ_WRITE) ==
143
             -1) {
           // error code already set by add_file_entry
144
           fat[first_block] = FAT_FREE;
145
146
           return -1:
147
148
149
         \ensuremath{//} fill in the file descriptor entry
150
         fd_table[fd].in_use = 1;
        fd_table[fd].ref_count++;
strncpy(fd_table[fd].filename, fname, 31);
fd_table[fd].filename[31] = '\0';
151
152
153
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
      return fd;
161 }
```

4.5.1.5 k read()

Kernel-level call to read a file.

Reads data from an open file.

Definition at line 166 of file fs_kfuncs.c.

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

```
182
        return -1;
183
       if (buf == NULL || n < 0) {
184
       P_ERRNO = P_EINVAL;
185
        return -1;
186
187
188
      if (n == 0) {
189
        return 0;
190
191
      // check if we're at EOF already
192
      if (fd_table[fd].position >= fd_table[fd].size) {
193
194
        return 0;
195
196
197
      \ensuremath{//} determine how many bytes we can actually read
      uint32_t bytes_to_read = n;
if (fd_table[fd].position + bytes_to_read > fd_table[fd].size) {
  bytes_to_read = fd_table[fd].size - fd_table[fd].position;
198
199
200
201
202
203
       \ensuremath{//} find the block containing the current position
      uint16_t current_block = fd_table[fd].first_block;
uint32_t block_index = fd_table[fd].position / block_size;
2.04
205
      uint32_t block_offset = fd_table[fd].position % block_size;
206
207
       \ensuremath{//} navigate to the correct block in the chain
208
      for (uint32_t i = 0; i < block_index; i++) {
   if (current_block == 0 || current_block == FAT_EOF) {</pre>
209
210
           // unexpected end of chain
211
           P_ERRNO = P_EINVAL;
212
213
           return -1;
214
215
         current_block = fat[current_block];
216
217
218
       // now we're at the right block, start reading
      uint32_t bytes_read = 0;
219
220
221
      while (bytes_read < bytes_to_read) {</pre>
222
         // how much data can we read from the current block
         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
                  ? (bytes_to_read - bytes_read)
226
227
                  : bytes_left_in_block;
228
229
         \ensuremath{//} seek to the right position in the file
         if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + block_offset,
230
                   SEEK_SET) == -1) {
231
           P_ERRNO = P_ELSEEK;
232
233
           if (bytes_read > 0) {
234
             fd_table[fd].position += bytes_read;
235
             return bytes_read;
236
237
           return -1;
238
239
         // read the data from the file
240
241
         ssize_t read_result = read(fs_fd, buf + bytes_read, bytes_to_read_now);
2.42
         if (read_result <= 0) {</pre>
          P_ERRNO = P_EREAD;
243
           // if we already read some data, return that count
if (bytes_read > 0) {
244
245
246
             fd_table[fd].position += bytes_read;
247
             return bytes_read;
248
249
           return -1:
250
251
252
         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>
           if (current_block == FAT_EOF) {
258
259
             // unexpected end of chain
260
             break;
261
           current_block = fat[current_block];
262
263
           block_offset = 0;
264
265
266
         // if we read less than expected, we might have hit EOF
2.67
         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 }
```

4.5.1.6 k_unlink()

```
int k_unlink ( \label{eq:const_char} \mbox{const_char} \ * \ \textit{fname} \ )
```

Kernel-level call to remove a file.

Removes a file from the file system.

Definition at line 566 of file fs_kfuncs.c.

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

4.5.1.7 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.

```
282
       // handle standard output and error
283
       if (fd == STDOUT_FILENO) {
        return write(STDOUT_FILENO, str, n);
284
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) {
   P_ERRNO = P_EINVAL;</pre>
295
296
297
        return -1;
298
299
      if (n == 0) {
300
        return 0;
301
302
      // check if filesystem is mounted and FAT is valid
if (!is_mounted || fat == NULL) {
303
304
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);
if (block_buffer == NULL) {
   P_ERRNO = P_EMALLOC;
314
315
316
        return -1;
317
318
319
320
      // calculate initial block position
      uint32_t block_index = current_position / block_size;
uint32_t block_offset = current_position % block_size;
321
322
323
324
       // if the file doesn't have a first block yet, allocate one
325
      if (current_block == 0) {
326
         current_block = allocate_block();
327
         if (current_block == 0) {
           P_ERRNO = P_EFULL;
328
           free (block_buffer);
329
330
           return -1;
331
332
         fd_table[fd].first_block = current_block;
333
334
335
       \ensuremath{//} navigate to the appropriate block
      uint16_t prev_block = 0;
336
337
       for (uint32_t i = 0; i < block_index; i++) {</pre>
338
        if (current_block == 0 || current_block == FAT_EOF ||
             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
           if (prev_block != 0 && prev_block < fat_size / 2) {</pre>
```

```
fat[prev_block] = new_block;
           } else {
  // if there's no previous block, this must be the first one
351
352
             fd_table[fd].first_block = new_block;
353
354
355
356
          current_block = new_block;
357
358
359
         prev_block = current_block;
360
         // validate the block number before accessing FAT
361
         if (current_block >= fat_size / 2) {
P_ERRNO = P_EINVAL;
362
363
364
           free(block_buffer);
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
      if (current_block == 0 || current_block == FAT_EOF ||
  current_block >= fat_size / 2) {
  if (prev_block != 0 && prev_block < fat_size / 2) {</pre>
372
373
374
375
          uint16_t new_block = allocate_block();
376
           if (new_block == 0) {
377
            P_ERRNO = P_EFULL;
378
             free(block_buffer);
379
             return -1;
380
381
382
           fat[prev_block] = new_block;
383
           current_block = new_block;
        } else {
  P_ERRNO = P_EINVAL;
384
385
           free (block_buffer);
386
387
           return -1;
388
389
390
      // start writing data
391
      uint32_t bytes_written = 0;
392
393
394
      while (bytes_written < n) {</pre>
395
         // validate current block
         if (current_block == 0 || current_block == FAT_EOF ||
    current_block >= fat_size / 2) {
396
397
           P_ERRNO = P_EINVAL;
398
399
           break:
400
401
402
         \ensuremath{//} how much can we write to this block
        uint32_t space_in_block = block_size - block_offset;
uint32_t bytes_to_write = (n - bytes_written) < space_in_block</pre>
403
404
                                           ? (n - bytes_written)
405
406
                                            : space_in_block;
407
408
         // position in filesystem
         off_t block_position = fat_size + (current_block - 1) * block_size;
409
410
         // if we're not writing a full block or not starting at the beginning, we
411
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
           \ensuremath{//} read the current block data
421
           ssize_t read_result = read(fs_fd, block_buffer, block_size);
422
           if (read_result < 0) {</pre>
             P_ERRNO = P_EREAD;
423
424
             break;
425
426
427
           \ensuremath{//} copy the new data into the block buffer
428
           memcpy(block_buffer + block_offset, str + bytes_written, bytes_to_write);
429
           // seek back to write the modified block
430
           if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
431
432
             P_ERRNO = P_ELSEEK;
433
             break;
434
           }
435
436
           // write the full block back
```

```
437
          ssize_t write_result = write(fs_fd, block_buffer, block_size);
438
          if (write_result != block_size) {
439
             P_ERRNO = P_EWRITE;
             // we might have a partial write, but that's hard to handle correctly
440
441
            break:
442
443
        } else {
444
          // we're writing a full block from the beginning
445
           if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
446
            P_ERRNO = P_ELSEEK;
447
            break:
448
449
450
          ssize_t write_result = write(fs_fd, str + bytes_written, bytes_to_write);
451
          if (write_result != bytes_to_write) {
452
            P_ERRNO = P_EWRITE;
453
            break;
454
          }
455
456
457
        // update counters
        bytes_written += bytes_to_write;
block_offset = (block_offset + bytes_to_write) % block_size;
458
459
460
461
        // if we've filled this block and still have more to write, go to the next
462
        // block
463
        if (block_offset == 0 && bytes_written < n) {</pre>
          // validate current block before accessing FAT
if (current_block >= fat_size / 2) {
   P_ERRNO = P_EINVAL;
464
465
466
467
            break:
468
469
470
           // check if there's a next block
471
          if (fat[current_block] == FAT_EOF) {
            // allocate a new block
472
            uint16_t new_block = allocate_block();
473
474
            if (new\_block == 0) {
475
              P_ERRNO = P_EFULL;
476
              break;
477
478
            // Update the FAT safely
if (current_block < fat_size / 2) {</pre>
479
480
              fat[current_block] = new_block;
482
483
              P_ERRNO = P_EINVAL;
484
              break;
            }
485
486
487
            current_block = new_block;
488
489
             current_block = fat[current_block];
490
       }
491
      }
492
493
494
      // free the block buffer
495
      free(block_buffer);
496
      // update file position
497
      fd_table[fd].position += bytes_written;
498
499
500
      // update file size if needed
501
      if (fd_table[fd].position > fd_table[fd].size) {
        fd_table[fd].size = fd_table[fd].position;
502
503
504
        // update the directory entry
        dir_entry_t entry;
505
506
        int dir_offset = find_file(fd_table[fd].filename, &entry);
507
        if (dir_offset >= 0) {
          entry.size = fd_table[fd].size;
508
          entry.mtime = time(NULL);
509
510
          if (lseek(fs fd, dir offset, SEEK SET) == -1) {
511
512
            P_ERRNO = P_ELSEEK;
513
            return -1;
514
          if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
515
516
            P ERRNO = P EWRITE;
            return -1;
517
518
519
520
521
522
      return bytes_written;
523 }
```

4.5.2 Variable Documentation

4.5.2.1 current_fg_pid

```
pid_t current_fg_pid [extern]
```

Definition at line 31 of file kern_sys_calls.c.

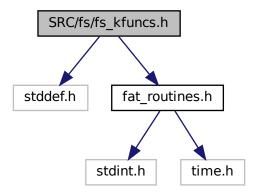
4.5.2.2 current_running_pcb

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

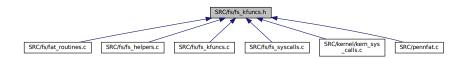
Definition at line 38 of file scheduler.c.

4.6 SRC/fs/fs_kfuncs.h File Reference

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



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



Macros

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

Functions

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

Opens a file with the specified mode.

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

Reads data from an open file.

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

Writes data to an open file.

int k_close (int fd)

Closes an open file.

• int k_unlink (const char *fname)

Removes a file from the file system.

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

Repositions the file offset of an open file.

• int k_ls (const char *filename)

Lists files or file information.

4.6.1 Macro Definition Documentation

4.6.1.1 SEEK_CUR

```
#define SEEK_CUR 1
```

Definition at line 17 of file fs_kfuncs.h.

4.6.1.2 SEEK_END

```
#define SEEK_END 2
```

Definition at line 18 of file fs_kfuncs.h.

4.6.1.3 SEEK_SET

```
#define SEEK_SET 0
```

Definition at line 16 of file fs_kfuncs.h.

4.6.2 Function Documentation

4.6.2.1 k_close()

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

Closes an open file.

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

Parameters

```
fd File descriptor of the open file.
```

Returns

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

• P_EBADF: Invalid file descriptor.

Closes an open file.

Definition at line 528 of file fs_kfuncs.c.

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

4.6.2.2 k_ls()

```
int k_ls ( \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 668 of file fs_kfuncs.c.
```

```
669
      if (!is mounted) {
        P_ERRNO = P_EFS_NOT_MOUNTED;
return -1;
670
671
672
673
674
      // start with root directory block
675
      uint16_t current_block = 1;
676
      dir_entry_t dir_entry;
      uint32_t offset = 0;
677
678
      // if filename is null, list all files in the current directory
680
      if (filename == NULL) {
681
        while (1) {
682
          \ensuremath{//} adjust pointer to beginning of current block
          if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) == -1) {
683
            P_ERRNO = P_ELSEEK;
684
685
            return -1;
686
687
          offset = 0;
688
689
690
          // search current block
691
          while (offset < block_size) {</pre>
692
            if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
693
               P_ERRNO = P_EREAD;
694
              return -1;
695
696
697
            // check if we've reached the end of directory
            if (dir_entry.name[0] == 0) {
699
              break;
700
701
            // skip deleted entries
if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
702
703
704
              offset += sizeof(dir_entry);
705
               continue;
706
707
708
            // format permission string
            char perm_str[4] = "---";
709
            if (dir_entry.perm & PERM_READ)
710
711
              perm_str[0] = 'r';
            if (dir_entry.perm & PERM_WRITE)
            perm_str[1] = 'w';
if (dir_entry.perm & PERM_EXEC)
perm_str[2] = 'x';
713
714
715
716
717
            // format time
```

```
718
            struct tm* tm_info = localtime(&dir_entry.mtime);
             char time_str[50];
719
             strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);
720
721
722
            // print entry details
723
            char buffer[128];
            int len;
724
725
            if (dir_entry.firstBlock == 0) {
             len = snprintf(buffer, sizeof(buffer), " -%s- %6d %s %s\n",
726
727
                       perm_str, dir_entry.size, time_str, dir_entry.name);
728
            } else {
             len = snprintf(buffer, sizeof(buffer), "%2d -%s- %6d %s %s\n",
dir_entry.firstBlock, perm_str, dir_entry.size, time_str, dir_entry.name);
729
730
731
732
733
            if (len < 0 || len >= (int)sizeof(buffer)) {
734
               P_ERRNO = P_EUNKNOWN;
              return -1;
735
736
737
738
             if (k_write(STDOUT_FILENO, buffer, len) != len) {
739
               P_ERRNO = P_EWRITE;
740
              return -1;
741
742
743
            offset += sizeof(dir_entry);
744
745
746
          \ensuremath{//} move to the next block if there is one
          if (fat[current_block] != FAT_EOF) {
747
748
            current block = fat[current block];
749
            continue;
750
751
752
           // no more blocks to search
753
          break:
754
755
      } else {
756
        // find and display specific file
        int file_offset = find_file(filename, &dir_entry);
if (file_offset < 0) {</pre>
757
758
         P_ERRNO = P_ENOENT;
759
          return -1;
760
761
762
763
        if (dir_entry.name[0] == 0) {
764
        P_ERRNO = P_ENOENT;
765
          return -1;
766
767
768
        // skip deleted entries
769
        if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
770
          P_ERRNO = P_ENOENT;
771
          return -1;
772
773
774
        // format permission string
775
        char perm_str[4] = "---"
        if (dir_entry.perm & PERM_READ)
  perm_str[0] = 'r';
776
777
778
        if (dir_entry.perm & PERM_WRITE)
        perm_str[1] = 'w';
if (dir_entry.perm & PERM_EXEC)
779
780
781
         perm_str[2] = 'x';
782
783
        // format time
784
        struct tm* tm_info = localtime(&dir_entry.mtime);
785
        char time str[50];
786
        strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);
787
788
        // print entry details
789
        char buffer[128];
790
        int len;
        if (dir_entry.firstBlock == 0) {
791
          len = snprintf(buffer, sizeof(buffer), " -%s- %6d %s %s\n",
792
793
                   perm_str, dir_entry.size, time_str, dir_entry.name);
794
795
         len = snprintf(buffer, sizeof(buffer), "%2d -%s- %6d %s %s\n",
796
                   dir_entry.firstBlock, perm_str, dir_entry.size, time_str, dir_entry.name);
797
798
799
        if (len < 0 || len >= (int)sizeof(buffer)) {
800
         P_ERRNO = P_EUNKNOWN;
801
          return -1;
802
803
        if (k_write(STDOUT_FILENO, buffer, len) != len) {
804
```

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.

```
// standard file descriptors don't support lseek
623
      if (fd == STDIN_FILENO || fd == STDOUT_FILENO || fd == STDERR_FILENO) {
   P_ERRNO = P_EINVAL;
624
625
        return -1;
626
627
628
629
      \ensuremath{//} validate the file descriptor
      if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
630
      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:
640
          new_position = offset;
```

```
641
         break;
642
       case SEEK_CUR:
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
654
     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
34
     if (fname == NULL || *fname == '\0') {
  P_ERRNO = P_EINVAL;
35
36
       return -1;
38
39
     if ((mode & (F_READ | F_WRITE | F_APPEND)) == 0) {
40
       P_ERRNO = P_EINVAL;
41
       return -1;
42
43
    // check if the file system is mounted
```

```
45
     if (!is_mounted) {
      P_ERRNO = P_EFS_NOT_MOUNTED;
46
47
        return -1;
     }
48
49
     // get a free file descriptor
50
     int fd = get_free_fd(fd_table);
51
     if (fd < 0) {</pre>
53
      P_ERRNO = P_EFULL; // no free file descriptors
54
        return -1;
     }
55
56
      // check if the file exists
     dir_entry_t entry;
59
     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
75
        // fill in the file descriptor entry
76
        fd_table[fd].in_use = 1;
77
        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;
80
        fd_table[fd].first_block = entry.firstBlock;
81
82
        fd_table[fd].mode = mode;
84
        // set the initial position
85
        if (mode & F_APPEND) {
         fd_table[fd].position = entry.size;
86
87
        } else {
88
          fd_table[fd].position = 0;
90
91
        // if mode includes F_WRITE and not F_APPEND, truncate the file
        if ((mode & F_WRITE) && !(mode & F_APPEND)) {
   // free all blocks except the first one
   uint16_t block = entry.firstBlock;
92
93
94
          uint16_t next_block;
95
96
97
          if (block != 0 && block != FAT_EOF) {
            next_block = fat[block];
fat[block] = FAT_EOF; /
98
                                       // terminate the chain at the first block
99
100
             block = next block;
101
102
              // 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 {\tt 0}
111
           fd_table[fd].size = 0;
112
           entry.size = 0;
113
           entry.mtime = time(NULL);
114
115
            // update the file system with the truncated file
116
           if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
             P_ERRNO = P_ELSEEK;
return -1;
117
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
         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();
        if (first_block == 0) {
  P_ERRNO = P_EFULL;
136
137
138
          return -1;
139
140
141
        // create a new file entry
        if (add_file_entry(fname, 0, first_block, TYPE_REGULAR, PERM_READ_WRITE) ==
142
143
           // 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;
150
151
        fd_table[fd].ref_count++;
152
        strncpy(fd_table[fd].filename, fname, 31);
153
        fd_table[fd].filename[31] = ' \setminus 0';
        fd_table[fd].size = 0;
fd_table[fd].first_block = first_block;
154
155
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.
buf	Buffer to store the read data.
n	Maximum number of bytes to read.

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.

```
s_kill(current_running_pcb->pid, P_SIGSTOP);
171
172
        }
173
      // handle standard input
174
175
      if (fd == STDIN_FILENO) {
176
        return read(STDIN_FILENO, buf, n);
177
178
      // validate inputs
if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
   P_ERRNO = P_EBADF;
179
180
181
        return -1;
182
183
      if (buf == NULL || n < 0) {
184
       P_ERRNO = P_EINVAL;
185
        return -1;
186
187
188
      if (n == 0) {
189
        return 0;
190
191
      // check if we're at EOF already
192
      if (fd_table[fd].position >= fd_table[fd].size) {
193
194
        return 0;
195
196
197
      \ensuremath{//} determine how many bytes we can actually read
      uint32_t bytes_to_read = n;
if (fd_table[fd].position + bytes_to_read > fd_table[fd].size) {
198
199
200
        bytes_to_read = fd_table[fd].size - fd_table[fd].position;
201
202
203
      \ensuremath{//} find the block containing the current position
      uint16_t current_block = fd_table[fd].first_block;
uint32_t block_index = fd_table[fd].position / block_size;
204
205
206
      uint32_t block_offset = fd_table[fd].position % block_size;
207
208
       // navigate to the correct block in the chain
      for (uint32_t i = 0; i < block_index; i++) {
   if (current_block == 0 || current_block == FAT_EOF) {</pre>
209
210
          // unexpected end of chain
P_ERRNO = P_EINVAL;
211
212
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)
225
226
227
                  : bytes_left_in_block;
228
229
        \ensuremath{//} seek to the right position in the file
        230
231
          P_ERRNO = P_ELSEEK;
232
233
          if (bytes_read > 0) {
234
             fd_table[fd].position += bytes_read;
235
            return bytes_read;
236
237
          return -1:
238
239
240
         \ensuremath{//} read the data from the file
241
         ssize_t read_result = read(fs_fd, buf + bytes_read, bytes_to_read_now);
        if (read_result <= 0) {</pre>
2.42
          P ERRNO = P EREAD:
243
          // if we already read some data, return that count
if (bytes_read > 0) {
244
245
246
             fd_table[fd].position += bytes_read;
247
             return bytes_read;
248
249
          return -1;
250
251
252
        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
```

```
if (block_offset == block_size && bytes_read < bytes_to_read) {</pre>
         if (current_block == FAT_EOF) {
259
            // unexpected end of chain
260
           break;
2.61
          current_block = fat[current_block];
262
         block_offset = 0;
263
264
265
266
        // if we read less than expected, we might have hit EOF
       if (read_result < bytes_to_read_now) {</pre>
267
268
         break;
269
270
271
272
273
      // update file position
     fd_table[fd].position += bytes_read;
274
     return bytes_read;
```

4.6.2.6 k_unlink()

```
int k_unlink ( \label{const_char} \mbox{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 | | *fname == ' \setminus 0') {
568
        P_ERRNO = P_EINVAL;
569
         return -1;
570
571
572
       if (!is mounted) {
573
        P_ERRNO = P_EFS_NOT_MOUNTED;
574
         return -1;
575
576
       // 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
580
          P_ERRNO = P_EBUSY;
581
            return -1;
        }
582
583
584
585 // find the file in directory
586 dir_entry_t entry;
```

```
587
      int file_offset = find_file(fname, &entry);
588
      if (file_offset < 0) {</pre>
589
        P_ERRNO = P_ENOENT;
590
        return -1;
591
592
593
      // mark the directory entry as deleted (set first byte to 1)
594
      entry.name[0] = 1;
595
596
      \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)) {
      P_ERRNO = P_EWRITE;
return -1;
602
603
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) {
  next_block = fat[current_block];
610
611
        fat[current_block] = FAT_FREE;
612
613
        current_block = next_block;
614
615
616
     return 0;
617 }
```

4.6.2.7 k write()

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

Writes data to an open file.

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

Parameters

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

Returns

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

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

Writes data to an open file.

Definition at line 281 of file fs_kfuncs.c.

```
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
      // validate inputs
290
      if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
  P_ERRNO = P_EBADF;
291
292
293
         return -1:
294
295
      if (str == NULL || n < 0) {
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) {
       P_ERRNO = P_EFS_NOT_MOUNTED;
305
306
         return -1;
307
308
309
      // get file information
      uint16_t current_block = fd_table[fd].first_block;
310
      uint32_t current_position = fd_table[fd].position;
311
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;
return -1;
316
317
318
319
320
       // calculate initial block position
      uint32_t block_index = current_position / block_size;
uint32_t block_offset = current_position % block_size;
321
322
323
      // if the file doesn't have a first block yet, allocate one
324
      if (current_block == 0) {
325
326
         current_block = allocate_block();
327
         if (current_block == 0) {
328
          P_ERRNO = P_EFULL;
           free(block_buffer);
329
330
           return -1:
331
332
         fd_table[fd].first_block = current_block;
333
334
      // navigate to the appropriate block
uint16_t prev_block = 0;
335
336
      for (uint32_t i = 0; i < block_index; i++) {</pre>
337
338
        if (current_block == 0 || current_block == FAT_EOF ||
339
             current_block >= fat_size / 2) {
           // reached the end of chain prematurely, need to allocate a new block
340
341
           uint16_t new_block = allocate_block();
           if (new_block == 0) {
  P_ERRNO = P_EFULL;
342
343
344
             free(block_buffer);
345
             return -1;
346
347
           // update the chain
if (prev_block != 0 && prev_block < fat_size / 2) {</pre>
348
349
350
             fat[prev_block] = new_block;
351
           } else {
352
             // if there's no previous block, this must be the first one
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
         if (current_block >= fat_size / 2) {
   P_ERRNO = P_EINVAL;
362
363
364
           free(block_buffer);
365
           return -1;
366
367
         current block = fat[current block];
368
```

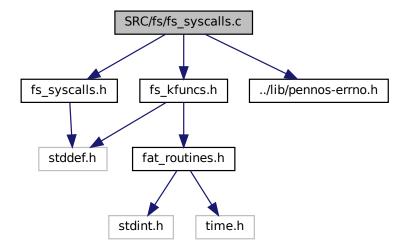
```
369
      }
370
371
      // if we ended up without a valid block, go back to the last valid one
      if (current_block == 0 || current_block == FAT_EOF ||
    current_block >= fat_size / 2) {
    if (prev_block != 0 && prev_block < fat_size / 2) {</pre>
372
373
374
375
          uint16_t new_block = allocate_block();
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;
383
           current_block = new_block;
384
        } else {
          P ERRNO = P EINVAL:
385
          free(block_buffer);
386
387
          return -1;
388
        }
389
      }
390
      // start writing data
391
      uint32_t bytes_written = 0;
392
393
394
      while (bytes_written < n) {</pre>
395
        // validate current block
        if (current_block == 0 || current_block == FAT_EOF ||
    current_block >= fat_size / 2) {
396
397
          P_ERRNO = P_EINVAL;
398
399
          break:
400
401
402
         // how much can we write to this block
        uint32_t space_in_block = block_size - block_offset;
uint32_t bytes_to_write = (n - bytes_written) < space_in_block</pre>
403
404
                                          ? (n - bytes_written)
405
406
                                          : space_in_block;
407
408
         // position in filesystem
409
        off_t block_position = fat_size + (current_block - 1) * block_size;
410
        // if we're not writing a full block or not starting at the beginning, we
411
        // need to read-modify-write
412
        if (bytes_to_write < block_size || block_offset > 0) {
413
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>
            P_ERRNO = P_EREAD;
423
424
            break;
425
426
427
           \ensuremath{//} copy the new data into the block buffer
428
          memcpy(block_buffer + block_offset, str + bytes_written, bytes_to_write);
429
           // seek back to write the modified block
430
431
          if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
           P_ERRNO = P_ELSEEK;
432
433
            break;
434
435
          // write the full block back
436
437
          ssize_t write_result = write(fs_fd, block_buffer, block_size);
438
           if (write_result != block_size) {
439
            P_ERRNO = P_EWRITE;
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
444
445
           if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
446
           P_ERRNO = P_ELSEEK;
447
            break;
          }
448
449
450
          ssize_t write_result = write(fs_fd, str + bytes_written, bytes_to_write);
451
          if (write_result != bytes_to_write) {
452
            P_ERRNO = P_EWRITE;
453
            break;
454
455
        }
```

```
456
         // update counters
457
         bytes_written += bytes_to_write;
block_offset = (block_offset + bytes_to_write) % block_size;
458
459
460
         // if we've filled this block and still have more to write, go to the next
461
462
463
         if (block_offset == 0 && bytes_written < n) {</pre>
464
          // validate current block before accessing FAT
           if (current_block >= fat_size / 2) {
   P_ERRNO = P_EINVAL;
465
466
467
             break:
468
469
470
           // check if there's a next block
471
           if (fat[current_block] == FAT_EOF) {
             // allocate a new block
472
            uint16_t new_block = allocate_block();
if (new_block == 0) {
473
474
475
               P_ERRNO = P_EFULL;
476
477
478
             // Update the FAT safely
if (current_block < fat_size / 2) {</pre>
479
480
               fat[current_block] = new_block;
482
483
               P_ERRNO = P_EINVAL;
484
               break;
485
486
487
             current_block = new_block;
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
501
      if (fd_table[fd].position > fd_table[fd].size) {
502
        fd_table[fd].size = fd_table[fd].position;
503
         // update the directory entry
504
         dir_entry_t entry;
505
        int dir_offset = find_file(fd_table[fd].filename, &entry);
if (dir_offset >= 0) {
506
507
           entry.size = fd_table[fd].size;
entry.mtime = time(NULL);
508
509
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;
             return -1;
517
518
           }
520
521
522
      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.

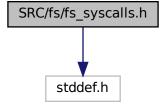
This is a wrapper around the kernel function k_write.

```
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
61    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.
```

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

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 remove.

Returns

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

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

Removes a file from the file system.

This is a wrapper around the kernel function k_unlink.

Definition at line 51 of file fs_syscalls.c.

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

4.8.2.7 s_write()

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

Writes data to an open file.

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

Parameters

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

Returns

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

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

Writes data to an open file.

This is a wrapper around the kernel function k_write.

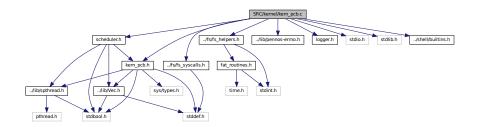
```
Definition at line 33 of file fs_syscalls.c.
```

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

4.9 SRC/kernel/kern_pcb.c File Reference

```
#include "kern_pcb.h"
#include "../fs/fs_helpers.h"
#include "../fs/fs_syscalls.h"
#include "../lib/pennos-errno.h"
#include "logger.h"
#include "scheduler.h"
#include "stdio.h"
#include "stdlib.h"
#include "../shell/builtins.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 42 of file kern_pcb.c.

```
47
     pcb_t* ret_pcb = malloc(sizeof(pcb_t));
48
     if (ret_pcb == NULL) {
49
     perror("malloc failed for PCB creation");
50
       return NULL;
    }
51
    ret_pcb->pid = pid;
    ret_pcb->par_pid = par_pid;
     ret_pcb->priority = priority;
55
                                       // running by default
    ret_pcb->process_state = 'R';
ret_pcb->input_fd = input_fd;
56
57
58
    ret_pcb->output_fd = output_fd;
    ret_pcb->process_status = 0; // default status
    ret_pcb->child_pcbs = vec_new(0, NULL); // NULL deconstructor prevents
61
                                                 // double free
62
63
    for (int i = 0; i < 3; i++) {
64
65
       ret_pcb->signals[i] = false;
    ret_pcb->is_sleeping = false;
ret_pcb->time_to_wake = -1; // default to not sleeping
68
69
70
     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 30 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 151 of file kern_pcb.c.

```
// if proc has parent (i.e. isn't init) then remove it from parent's child
152
153
     pcb_t* par_pcb = get_pcb_in_queue(&current_pcbs, proc->par_pid);
154
     if (par_pcb != NULL) {
155
       remove_child_in_parent(par_pcb, proc);
156
     } else {
157
158
       P_ERRNO = P_ENULL;
159
       return;
160
161
162
     // if proc has children, remove them and assign them to init parent
     if (vec_len(&proc->child_pcbs) > 0) {
163
       // retrieve the init process
164
165
       pcb_t* init_pcb =
           get_pcb_in_queue(&current_pcbs, 1); // init process has pid 1
166
167
168
       while (vec_len(&proc->child_pcbs) > 0) {
         pcb_t* curr_child = vec_get(&proc->child_pcbs, 0);
169
170
          vec_push_back(&init_pcb->child_pcbs, curr_child);
171
         vec_erase_no_deletor(&proc->child_pcbs, 0); // don't free in erase
         172
173
174
175
176
177
      // decr reference counts + close files if necessary
178
     for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
  if (proc->fd_table[i] != -1) {
179
180
            (decrement_fd_ref_count(proc->fd_table[i]) == 0) {
181
182
           if (s_close(proc->fd_table[i]) == -1) {
183
             u_perror("closing on a non-valid fd");
184
           }
185
         }
186
       }
187
     }
```

```
189
      // cancel + join this thread
190
      spthread_cancel(proc->thread_handle);
191
      spthread_continue(proc->thread_handle);
192
      spthread_suspend(proc->thread_handle);
193
      spthread_join(proc->thread_handle, NULL);
194
195
      // delete this process from any queue it's in + free it
196
      delete_process_from_all_queues(proc);
197 free_pcb(proc);
198 }
```

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 94 of file kern_pcb.c.

```
95
     if (parent == NULL) { // init creation case
96
       pcb_t* init = create_pcb(1, 0, 0, 0, 1);
       if (init == NULL) {
  P_ERRNO = P_ENULL;
97
98
         return NULL;
99
100
101
        init->fd_table[0] = STDIN_FILENO;
102
        init->fd_table[1] = STDOUT_FILENO;
        init->fd_table[2] = STDERR_FILENO;
103
104
        for (int i = 3; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {</pre>
          init->fd_table[i] = -1;
106
107
108
        increment_fd_ref_count(STDIN_FILENO);
109
        increment_fd_ref_count(STDOUT_FILENO);
110
        increment_fd_ref_count(STDERR_FILENO);
111
112
        current running pcb = init;
113
        put_pcb_into_correct_queue(init);
114
        vec_push_back(&current_pcbs, init);
115
        return init;
116
117
      pcb_t* child = create_pcb(next_pid++, parent->pid, priority, parent->input_fd,
118
                                 parent->output_fd);
119
120
      if (child == NULL) {
121
       P_ERRNO = P_ENULL;
122
        return NULL;
123
124
125
      // copy parent's fd table
126
      for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {</pre>
127
       child->fd_table[i] = parent->fd_table[i];
128
129
      for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
  if (child->fd_table[i] != -1) {
130
131
132
          increment_fd_ref_count(child->fd_table[i]);
133
134
135
      // update parent as needed
136
      vec_push_back(&parent->child_pcbs, child);
137
138
139
      // add to appropriate queue
140
      put_pcb_into_correct_queue(child);
141
      vec_push_back(&current_pcbs, child);
142
143
      return child;
144 }
```

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 77 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;
   }
}

// Property of the property of
```

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.

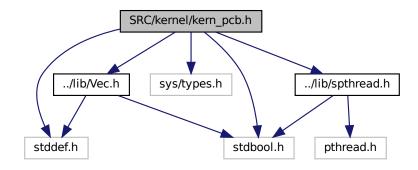
4.9.2.3 next_pid

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

Definition at line 17 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 42 of file kern_pcb.c.

```
pcb_t* ret_pcb = malloc(sizeof(pcb_t));
47
      if (ret_pcb == NULL) {
  perror("malloc failed for PCB creation");
48
         return NULL;
51
52
     ret_pcb->pid = pid;
ret_pcb->par_pid = par_pid;
ret_pcb->priority = priority;
5.3
54
     ret_pcb->process_state = 'R'
                                                // running by default
57
      ret_pcb->input_fd = input_fd;
58
     ret_pcb->output_fd = output_fd;
      ret_pcb->process_status = 0; // default status
59
60
     ret_pcb->child_pcbs = vec_new(0, NULL); // NULL deconstructor prevents
61
conc i = v; i < 3; i++) {
form ret_pcb->signals[i] = false;
form ret_pcb->signals[i] = false;
form ret_pcb->signals[i] = false;
form ret_pcb->signals[i] = false;
     for (int i = 0; i < 3; i++) {</pre>
     ret_pcb->is_sleeping = false;
     ret_pcb->time_to_wake = -1; // default to not sleeping
70
71
      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 30 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 151 of file kern_pcb.c.

```
151
      // if proc has parent (i.e. isn't init) then remove it from parent's child
153
      // list
      pcb_t* par_pcb = get_pcb_in_queue(&current_pcbs, proc->par_pid);
154
      if (par_pcb != NULL) {
155
       remove_child_in_parent(par_pcb, proc);
156
     } else {
157
      P_ERRNO = P_ENULL;
158
159
        return;
160
161
162
      // if proc has children, remove them and assign them to init parent
     if (vec_len(&proc->child_pcbs) > 0) {
   // retrieve the init process
163
165
       pcb_t* init_pcb =
166
            get_pcb_in_queue(&current_pcbs, 1); // init process has pid 1
167
       while (vec_len(&proc->child_pcbs) > 0) {
  pcb_t* curr_child = vec_get(&proc->child_pcbs, 0);
168
169
170
          vec_push_back(&init_pcb->child_pcbs, curr_child);
171
          vec_erase_no_deletor(&proc->child_pcbs, 0); // don't free in erase
          172
173
174
175
      }
177
178
      // decr reference counts + close files if necessary
179
      for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {</pre>
       if (proc->fd_table[i] != -1) {
180
         if (decrement_fd_ref_count(proc->fd_table[i]) == 0) {
181
           if (s_close(proc->fd_table[i]) == -1) {
182
183
              u_perror("closing on a non-valid fd");
184
185
          }
       }
186
187
188
      // cancel + join this thread
189
190
      spthread_cancel(proc->thread_handle);
191
      spthread_continue(proc->thread_handle);
192
      spthread_suspend(proc->thread_handle);
193
      spthread_join(proc->thread_handle, NULL);
194
195
      // delete this process from any queue it's in + free it
```

```
196    delete_process_from_all_queues(proc);
197    free_pcb(proc);
198 }
```

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

```
134
     }
135
136
     // update parent as needed
137
     vec_push_back(&parent->child_pcbs, child);
138
139
     // add to appropriate queue
140
     put_pcb_into_correct_queue(child);
141
     vec_push_back(&current_pcbs, child);
142
143
     return child;
144 }
```

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 77 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;
   }
}

}

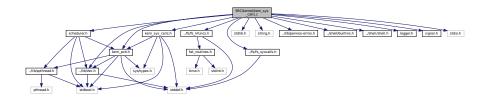
**Body
**The control of the contr
```

4.11 SRC/kernel/kern_sys_calls.c File Reference

```
#include "kern_sys_calls.h"
#include <stdlib.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
    if (prev_priority == 0) {
60
      prev_queue = &zero_priority_queue;
61
   } else if (prev_priority == 1) {
      prev_queue = &one_priority_queue;
64
65
      prev_queue = &two_priority_queue;
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 }
```

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;
360
     if (argv[1] == NULL) { // no args case
361
     s_exit();
362
       return NULL;
363
364
     365
366
367
      if (s_write(current_running_pcb->output_fd, argv[i], strlen(argv[i])) == -1) {
        u_perror("s_write error");
368
369
370
      if (s_write(current_running_pcb->output_fd, " ", 1) == -1) {
371
       u_perror("s_write error");
372
373
374
375
376
377 u_perror("s_write error");
378 }
     if (s_write(current_running_pcb->output_fd, "\n", 1) == -1) {
379
     return NULL;
380 }
```

4.11.1.8 s exit()

```
void s_exit (
          void )
```

Exits the current process and cleans up its resources.

Unconditionally exit the calling process.

Definition at line 293 of file kern_sys_calls.c.

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

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;
323
         return 0;
324
325
326
      return -1; // pid not found
```

4.11.1.11 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 385 of file kern_sys_calls.c.

```
char pid_top[] = "PID\tPPID\tPRI\tSTAT\tCMD\n";
386
387
     if (s_write(current_running_pcb->output_fd, pid_top, strlen(pid_top)) == -1) {
      u_perror("s_write error");
388
389
390
     for (int i = 0; i < vec_len(&current_pcbs); i++) {</pre>
     pcb_t* curr_pcb = (pcb_t*)vec_get(&current_pcbs, i);
391
392
      char buffer[100];
      393
394
395
              curr_pcb->cmd_str);
396
      if (s_write(current_running_pcb->output_fd, buffer, strlen(buffer)) == -1) {
397
        u_perror("s_write error");
     }
398
399
    return NULL;
400
401 }
```

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:
178
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);
227
            k_proc_cleanup(child);
```

```
228
             return child->pid;
229
230
231
        // If nohang is true, return immediately if no child has exited
2.32
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
               \label{log_generic_event} $$\log_{\mathtt{generic_event}}('\mathtt{W}', \mathtt{child->pid}, \mathtt{child->priority}, \mathtt{child->cmd\_str})$;
2.50
               vec_erase_no_deletor(&zombie_queue, i);
delete_from_explicit_queue(&parent->child_pcbs, child->pid);
251
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.

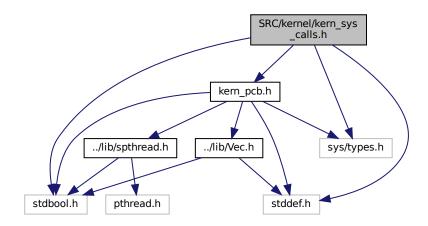
4.11.2.9 zombie_queue

```
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_t s_spawn_init ()

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

void s cleanup init process ()

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

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

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

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

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

• int s_kill (pid_t pid, int signal)

Send a signal to a particular process.

void s_exit (void)

Unconditionally exit the calling process.

• int s_nice (pid_t pid, int priority)

Set the priority of the specified thread.

void s_sleep (unsigned int ticks)

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

void * s_echo (void *arg)

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

void * s_ps (void *arg)

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

4.12.1 Function Documentation

4.12.1.1 delete_from_explicit_queue()

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.

```
89
     Vec* queue = NULL;
    if (queue_id == 0) {
91
      queue = &zero_priority_queue;
   } else if (queue_id == 1) {
92
93
      queue = &one_priority_queue;
   } else {
94
     queue = &two_priority_queue;
95
98 int index = determine_index_in_queue(queue, pid);
    if (index != -1) {
99
100
       vec_erase_no_deletor(queue, index);
101
102 }
```

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.

```
40
41     for (int i = 0; i < vec_len(queue); i++) {
42         pcb_t* curr_pcb = vec_get(queue, i);
43         if (curr_pcb->pid == pid) {
44             return i;
45         }
46         }
47
48         return -1; // not found
49 }
```

4.12.1.4 init_func()

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};
119
      s_spawn(shell, shell_argv, STDIN_FILENO, STDOUT_FILENO);
120
     // continuously wait for and reap zombie children
121
122
     while (true) {
123
      int status;
     s_waitpid(-1, &status, false);
}
124
125
126
     return NULL; // should never reach
127
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.

```
56
      Vec* prev_queue;
      Vec* new_queue;
59
60
     if (prev_priority == 0) {
    prev_queue = &zero_priority_queue;
} else if (prev_priority == 1) {
  prev_queue = &one_priority_queue;
61
62
63
       prev_queue = &two_priority_queue;
66
67
     if (new_priority == 0) {
  new_queue = &zero_priority_queue;
68
69
    } else if (new_priority == 1) {
70
        new_queue = &one_priority_queue;
72
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);
76
     if (ind != -1) {
79
        vec_erase_no_deletor(prev_queue, ind);
8.0
81
82
     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()

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;
359
     if (argv[1] == NULL) { // no args case
360
361
     s_exit();
      return NULL;
362
363
     364
365
366
367
368
369
      if (s_write(current_running_pcb->output_fd, " ", 1) == -1) {
370
      _ ,o_wilte(current_running_
u_perror("s_write error");
}
371
372
373
     }
374
375
376
     if (s_write(current_running_pcb->output_fd, "\n", 1) == -1) {
377    u_perror("s_write error");
378 }
379
    return NULL;
380 }
```

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
      current_running_pcb->process_state = 'Z';
295
296
      current_running_pcb->process_status = 20; // EXITED_NORMALLY
297
298
      // Log the exit
299
      log_generic_event('E', current_running_pcb->pid,
300
                        current_running_pcb->priority,
301
                        current_running_pcb->cmd_str);
302
303
     delete_from_queue(current_running_pcb->priority, current_running_pcb->pid);
304
     log_generic_event('Z', current_running_pcb->pid,
305
306
                        current_running_pcb->priority,
                        current_running_pcb->cmd_str);
307
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 385 of file kern_sys_calls.c.

```
char pid_top[] = "PID\tPPID\tPRI\tSTAT\tCMD\n";
386
387
     if (s_write(current_running_pcb->output_fd, pid_top, strlen(pid_top)) == -1) {
       u_perror("s_write error");
388
389
     for (int i = 0; i < vec_len(&current_pcbs); i++) {</pre>
391
      pcb_t* curr_pcb = (pcb_t*)vec_get(&current_pcbs, i);
392
        char buffer[100];
393
       snprintf(buffer, sizeof(buffer), "%d\t%d\t%d\t%c\t%s\n", curr_pcb->pid,
394
                 curr_pcb->par_pid, curr_pcb->priority, curr_pcb->process_state,
                curr_pcb->cmd_str);
395
      if (s_write(current_running_pcb->output_fd, buffer, strlen(buffer)) == -1) {
396
397
         u_perror("s_write error");
      }
398
399
400
    return NULL;
401 }
```

4.12.1.12 s_sleep()

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

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

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

Parameters

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

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

Definition at line 332 of file kern_sys_calls.c.

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

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.

```
164
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
172
      if (child == NULL) {
173
       P_ERRNO = P_ENULL;
174
        return -1;
175
176
177
178
      spthread_t thread_handle;
      if (spthread_create(&thread_handle, NULL, func, argv) != 0) {
179
180
       perror("Error in spthread_create in s_spawn call");
181
182
183
      child \rightarrow cmd\_str = strdup(argv[0]);
184
      child->thread_handle = thread_handle;
      child->input_fd = fd0;
child->output_fd = fd1;
185
186
187
      child->fd_table[0] = fd0;
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.

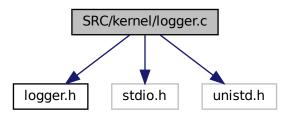
Definition at line 198 of file kern_sys_calls.c.

```
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} $$\log_{\mathtt{generic_event}}('\mathtt{W'}, \mathtt{child->pid}, \mathtt{child->priority}, \mathtt{child->cmd\_str})$;
225
           vec_erase_no_deletor(&zombie_queue, i);
226
            delete_from_explicit_queue(&parent->child_pcbs, child->pid);
227
           k_proc_cleanup(child);
228
           return child->pid;
229
230
231
232
       // If nohang is true, return immediately if no child has exited
       if (nohang) {
233
234
        return 0;
235
236
237
       // Block the parent until a child exits
      delete_from_queue(parent->priority, parent->pid);
parent->process_state = 'B';
log_generic_event('B', parent->pid, parent->priority, parent->cmd_str);
238
239
240
241
242
       while (true) {
243
          // Scan the zombie queue first for terminated children.
244
          for (int i = 0; i < vec_len(&zombie_queue); i++) {</pre>
           pcb_t* child = vec_get(&zombie_queue, i);
if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
245
246
247
             if (wstatus != NULL) {
248
                *wstatus = child->process_status;
249
250
              \label{log_generic_event} $$\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];

f(%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()

```
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

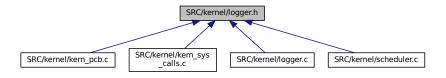
Postcondition

will perror if the write fails

Definition at line 6 of file logger.c.

4.14 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

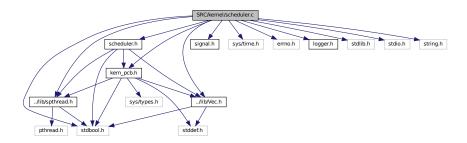
```
int tick_counter [extern]
```

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

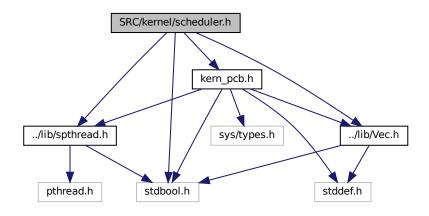
```
Vec zombie_queue
```

Definition at line 27 of file scheduler.c.

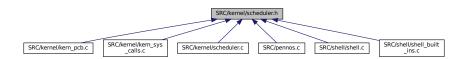
4.16 SRC/kernel/scheduler.h File Reference

```
#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()

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.

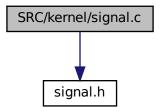
```
int curr_priority_queue_num;
280
      // mask for while scheduler is waiting for alarm
281
      sigset_t suspend_set;
282
      sigfillset(&suspend_set);
      sigdelset(&suspend_set, SIGALRM);
283
284
285
      // ensure sigarlm doesn't terminate the process
      struct sigaction act = (struct 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

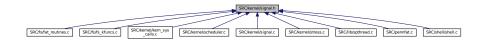
#include <signal.h>

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.

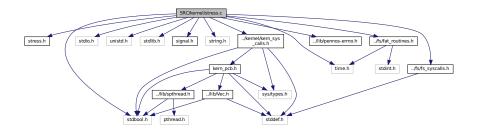
4.18.1.10 TERM_BY_SIG

```
#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()

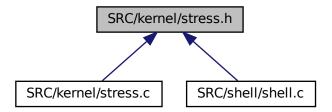
```
void* nohang (
     void * arg )
```

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 }
```

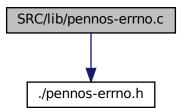
4.20.1.4 recur()

```
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_EREDIR 24
- #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_EREDIR

```
#define P_EREDIR 24
```

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

4.22.1.21 P_ESIGNAL

```
#define P_ESIGNAL 17
```

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

4.22.1.22 P EUNKNOWN

```
#define P_EUNKNOWN 99
```

Definition at line 31 of file pennos-errno.h.

4.22.1.23 P_EWRITE

```
#define P_EWRITE 18
```

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

4.22.1.24 P_INITFAIL

```
#define P_INITFAIL 23
```

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

4.22.1.25 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

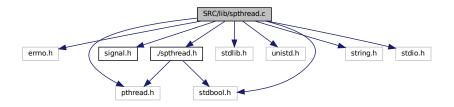
```
int P_ERRNO [extern]
```

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

```
{\tt 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()

Definition at line 293 of file spthread.c.

```
return pthread_cancel(thread.thread);
295 }
```

4.23.3.2 spthread_continue()

```
int spthread_continue (
               spthread_t thread )
Definition at line 241 of file spthread.c.
241
242
      pthread_t pself = pthread_self();
243
244
      if (pthread_equal(pself, thread.thread) != 0) {
       // 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) {
          .signal = SPTHREAD_SIG_CONTINUE,
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
      if (ret != 0) {
260
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
268
      // setting up args to nanosleep
269
      const struct timespec t = (struct timespec) {
          .tv_nsec = MILISEC_IN_NANO,
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
280
        // fprintf(stderr, "susp checking...\n");
281
        pthread_mutex_lock(&args.shutup_mutex);
282
        if (thread.meta->state == SPTHREAD_TERMINATED_STATE) {
283
         // child called exit, can break
284
285
          break;
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_fwd_args* fwd_args = malloc(sizeof(spthread_fwd_args));
123
124
      if (fwd_args == NULL) {
125
        free(child_meta);
126
       return EAGAIN;
127
128
      *fwd args = (spthread fwd args) {
         .actual_routine = start_routine,
129
130
          .actual_arg = arg,
131
          .setup_done = false
          .child_meta = child_meta,
132
     };
133
134
     int ret = pthread_mutex_init(&(fwd_args->setup_mutex), NULL);
if (ret != 0) {
135
136
137
       free(child_meta);
138
        free(fwd_args);
139
        return EAGAIN;
140
141
142
      ret = pthread_cond_init(&(fwd_args->setup_cond), NULL);
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));
154
      while (fwd_args->setup_done == false) {
155
       pthread_cond_wait(&(fwd_args->setup_cond), &(fwd_args->setup_mutex));
156
157
      pthread_mutex_unlock(&(fwd_args->setup_mutex));
158
      pthread_cond_destroy(&(fwd_args->setup_cond));
159
160
      pthread_mutex_destroy(&(fwd_args->setup_mutex));
161
      free(fwd_args);
162
163
      *thread = (spthread_t) {
164
       .thread = pthread,
          .meta = child_meta,
165
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);
if (res != 0) {
329
330
        return res;
331
332
      res = sigaddset(&block_set, SIGPTHD);
333
      if (res != 0) {
334
       return res;
335
      res = pthread_sigmask(SIG_BLOCK, &block_set, NULL);
336
337
      if (res != 0) {
338
       return res;
339
340
      return 0;
341 }
```

4.23.3.5 spthread_enable_interrupts_self()

```
int spthread_enable_interrupts_self ( )
Definition at line 345 of file spthread.c.
      sigset_t block_set;
      int res = sigemptyset(&block_set);
if (res != 0) {
347
348
349
       return res;
350
351
      res = sigaddset(&block_set, SIGPTHD);
      if (res != 0) {
353
354
355
     res = pthread_sigmask(SIG_UNBLOCK, &block_set, NULL);
vee != 0)
ceturn res;
358 }
350
     if (res != 0) {
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()

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()

```
bool spthread_self (
               spthread_t * thread )
Definition at line 297 of file spthread.c.
297
      if (my_meta == NULL) {
298
299
      return false;
300
301
      *thread = (spthread_t) {
      .thread = pthread_self(),
302
303
         .meta = my_meta,
304 };
305    return true;
306 }
```

4.23.3.10 spthread_suspend()

Definition at line 171 of file spthread.c.

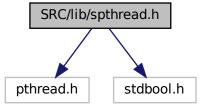
```
171
172
      pthread_t pself = pthread_self();
173
174
      if (pthread_equal(pself, thread.thread) != 0) {
175
        return spthread_suspend_self();
176
177
178
      spthread_signal_args args = (spthread_signal_args) {
179
          .signal = SPTHREAD_SIG_SUSPEND,
          .ack = 0,
180
181
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) {
       pthread_mutex_destroy(&args.shutup_mutex);
189
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) {
196
197
198
          .tv_nsec = MILISEC_IN_NANO,
199
200
201
      nanosleep(&t, NULL);
202
      pthread_mutex_lock(&args.shutup_mutex);
203
      while (args.ack != 1) {
   // wait for a mili second
204
205
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
        if (thread.meta->state == SPTHREAD_TERMINATED_STATE) {
214
215
          // child called exit, can break
216
          break;
217
        }
218
219
      pthread_mutex_unlock(&args.shutup_mutex);
220
221
      pthread_mutex_destroy(&args.shutup_mutex);
222
      return ret:
```

4.23.3.11 spthread_suspend_self()

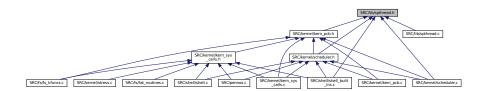
```
int spthread\_suspend\_self ( )
Definition at line 225 of file spthread.c.
        spthread_t self;
226
       bool am_sp = spthread_self(&self);
if (!am_sp) {
  return ESRCH;
227
228
229
230
231
       my_meta->state = SPTHREAD_SUSPENDED_STATE;
233
234
       sigsuspend(&my_meta->suspend_set);
}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

4.24 SRC/lib/spthread.h File Reference

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



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



Classes

• struct spthread_st

Macros

• #define SIGPTHD SIGUSR1

Typedefs

- typedef struct spthread_meta_st spthread_meta_t
- · typedef struct spthread st spthread t

Functions

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

4.24.1 Macro Definition Documentation

4.24.1.1 SIGPTHD

#define SIGPTHD SIGUSR1

Definition at line 19 of file spthread.h.

4.24.2 Typedef Documentation

4.24.2.1 spthread_meta_t

typedef struct spthread_meta_st spthread_meta_t

Definition at line 1 of file spthread.h.

4.24.2.2 spthread_t

```
typedef struct spthread_st spthread_t
```

4.24.3 Function Documentation

4.24.3.1 spthread_cancel()

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 runnning... 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) {
           .signal = SPTHREAD_SIG_CONTINUE,
251
           .ack = 0,
252
253
      pthread_mutex_init(&args.shutup_mutex, NULL);
254
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
      // wait for our signal to be ack'd
266
267
      // setting up args to nanosleep
const struct timespec t = (struct timespec) {
268
269
270
271
         .tv_nsec = MILISEC_IN_NANO,
272
273
      pthread_mutex_lock(&args.shutup_mutex);
274
      while (args.ack != 1) {
       // wait for a mili second
275
276
        pthread_mutex_unlock(&args.shutup_mutex);
277
278
        nanosleep(&t, NULL);
279
280
        // fprintf(stderr, "susp checking...\n");
        pthread_mutex_lock(&args.shutup_mutex);
```

4.24.3.3 spthread_create()

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

4.24.3.4 spthread_disable_interrupts_self()

```
\verb|int spthread_disable_interrupts_self ()|\\
```

Definition at line 326 of file spthread.c.

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

```
346
      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
     return res;
}
      if (res != 0) {
353
354
355
     res = pthread_sigmask(SIG_UNBLOCK, &block_set, NULL);
     , tes != 0)
return res;
}
     if (res != 0) {
357
358
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.
316 // necessary cleanup is registered
317 // in a cleanup routine
318 // that is pushed at start of an spthread
319 pthread_exit(status);
320 }
```

4.24.3.8 spthread_join()

```
int spthread_join (
            spthread_t thread,
            void ** retval )
```

Definition at line 308 of file spthread.c.

```
int res = pthread_join(thread.thread, retval);
int res = pthread_mutex_destroy(&thread.meta->meta_mutex);
int res = pthread_mutex_destroy(&thread.meta->meta_mutex);
int res = pthread_mutex_destroy(&thread.meta->meta_mutex);
int res = pthread_mutex_destroy(&thread.meta->meta_mutex);
int return res;
int return
```

4.24.3.9 spthread_self()

```
bool spthread_self (
           spthread_t * thread )
```

Definition at line 297 of file spthread.c.

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

4.24.3.10 spthread_suspend()

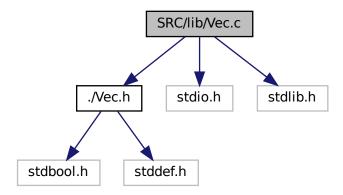
```
int spthread_suspend (
                spthread_t thread )
Definition at line 171 of file spthread.c.
171
172
      pthread_t pself = pthread_self();
173
174
      if (pthread_equal(pself, thread.thread) != 0) {
      return spthread_suspend_self();
}
175
176
177
      spthread_signal_args args = (spthread_signal_args) {
    .signal = SPTHREAD_SIG_SUSPEND,
178
179
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);
      while (args.ack != 1) {
   // wait for a mili second
204
205
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 }
```

4.24.3.11 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
     my_meta->state = SPTHREAD_SUSPENDED_STATE;
232
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()

```
void vec_destroy ( \label{eq:vec_destroy} \mbox{Vec} \ * \ self \ )
```

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

Parameters

self a pointer to the vector we want to destruct.

Precondition

Assumes self points to a valid vector.

Postcondition

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

Definition at line 17 of file Vec.c.

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.
index	the index of the element we want to erase at. Elements after this index are "shifted" down one position.

Precondition

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

Definition at line 63 of file Vec.c.

```
63
64    if (index >= self->length) {
65        perror("vec_erase: index >= vec length");
66    }
67
68    if (self->ele_dtor_fn) {
69        self->ele_dtor_fn(self->data[index]);
70    }
71
72    for (unsigned int i = index; i < self->length - 1; i++) {
73        self->data[i] = self->data[i + 1];
74    }
75
76    self->length--;
77 }
```

4.25.1.4 vec_erase_no_deletor()

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

Parameters

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

Definition at line 79 of file Vec.c.

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

```
88 self->length--;
89 }
```

4.25.1.5 vec_get()

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) {
92
93
       perror("vec_insert: index > vec length");
96
    if (index == self->length) { // Insertion at end = Adding at end
97
        vec_push_back(self, new_ele);
    } else { // Inserting not at the end // Vector is full
98
99
100
        if (self->length == self->capacity) {
101
           vec_resize(self, self->capacity * 2);
102
        // Insertion + Displacement
for (unsigned int i = self->length; i > index; i--) {
  self->data[i] = self->data[i - 1];
103
104
105
106
107
        self->data[index] = new_ele;
108
109
        self->length++;
110 }
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.

```
125
126
        if (self->capacity == self->length) {
  if (self->capacity == 0) {
    vec_resize(self, 1);
127
129
130
             vec_resize(self, self->capacity * 2);
131
132
133
134
        if (self->capacity == self->length) {
         perror("vec_push_back: resize failed");
136
137
       // The array is 0 indexed
self->data[self->length++] = new_ele;
138
139
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.

```
45
      if (new_capacity * sizeof(void*) < new_capacity) {</pre>
46
       perror("vec_resize: new capacity too large");
     if (new_capacity > self->length) {
  self->capacity = new_capacity;
48
49
        ptr_t* new_data = malloc(sizeof(void*) * self->capacity);
50
51
       // Copy over old elements
for (int i = 0; i < self->length; i++) {
52
54
         new_data[i] = self->data[i];
55
56
       free(self->data);
57
58
       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

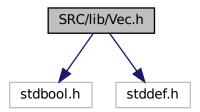
Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

Definition at line 142 of file Vec.c.

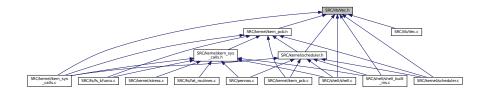
```
142
143     if (index >= self->length) {
144         perror("vec_set: idx >= len");
145     }
146     if (self->ele_dtor_fn) {
147         self->ele_dtor_fn(self->data[index]);
148     }
149     self->data[index] = new_ele;
150 }
```

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)
```

4.26.1 Macro Definition Documentation

void vec_destroy (Vec *self)

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 \ ) \ (({\tt vec}) -> {\tt length})
```

Returns the current length of the Vec written as a function-like macro

Parameters

vec,a pointer to the vector we want to grab the len of.

Definition at line 47 of file Vec.h.

4.26.2 Typedef Documentation

4.26.2.1 ptr_dtor_fn

```
typedef void(* ptr_dtor_fn) (ptr_t)
```

Definition at line 8 of file Vec.h.

4.26.2.2 ptr_t

```
typedef void* ptr_t
```

Definition at line 7 of file Vec.h.

4.26.2.3 Vec

```
typedef struct vec_st Vec
```

4.26.3 Function Documentation

4.26.3.1 vec_clear()

```
void vec_clear ( \label{eq:vec_vec} \mbox{Vec} \ * \ self \ )
```

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

Parameters

self a pointer to the vector we want to clear.

Precondition

Assumes self points to a valid vector.

Postcondition

The removed elements are destructed (cleaned up).

Definition at line 34 of file Vec.c.

4.26.3.2 vec destroy()

```
void vec_destroy ( \label{eq:vec_destroy} \mbox{Vec} \ * \ self \ )
```

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

Parameters

self a pointer to the vector we want to destruct.

Precondition

Assumes self points to a valid vector.

Postcondition

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

Definition at line 17 of file Vec.c.

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.
index	the index of the element we want to erase at. Elements after this index are "shifted" down one position.

Precondition

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

Definition at line 63 of file Vec.c.

```
63
64    if (index >= self->length) {
65        perror("vec_erase: index >= vec length");
66    }
67
68    if (self->ele_dtor_fn) {
69        self->ele_dtor_fn(self->data[index]);
70    }
71
72    for (unsigned int i = index; i < self->length - 1; i++) {
73        self->data[i] = self->data[i + 1];
74    }
75
76    self->length--;
77 }
```

4.26.3.4 vec_erase_no_deletor()

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

Parameters

se	elf	a pointer to the vector we want to erase from
in	dex	the index of the element we want to erase at

Definition at line 79 of file Vec.c.

```
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) {
92
93
       perror("vec_insert: index > vec length");
96
    if (index == self->length) { // Insertion at end = Adding at end
97
        vec_push_back(self, new_ele);
    } else { // Inserting not at the end
// Vector is full
98
99
100
        if (self->length == self->capacity) {
101
           vec_resize(self, self->capacity * 2);
102
        // Insertion + Displacement
for (unsigned int i = self->length; i > index; i--) {
  self->data[i] = self->data[i - 1];
103
104
105
106
107
        self->data[index] = new_ele;
108
109
        self->length++;
110 }
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.

```
125
126
        if (self->capacity == self->length) {
  if (self->capacity == 0) {
    vec_resize(self, 1);
127
129
130
             vec_resize(self, self->capacity * 2);
131
132
133
134
        if (self->capacity == self->length) {
         perror("vec_push_back: resize failed");
136
137
       // The array is 0 indexed
self->data[self->length++] = new_ele;
138
139
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.

```
45
      if (new_capacity * sizeof(void*) < new_capacity) {</pre>
46
       perror("vec_resize: new capacity too large");
     if (new_capacity > self->length) {
  self->capacity = new_capacity;
48
49
       ptr_t* new_data = malloc(sizeof(void*) * self->capacity);
50
51
       // Copy over old elements
for (int i = 0; i < self->length; i++) {
52
54
         new_data[i] = self->data[i];
55
56
      free(self->data);
57
58
       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

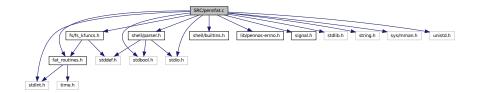
Definition at line 142 of file Vec.c.

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

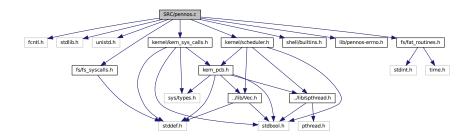
4.27.2.1 main()

```
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.

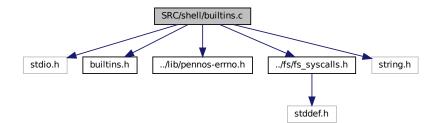
4.28.2.2 tick_counter

```
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)

Creates a user-level error message similar to perror.

4.29.1 Function Documentation

4.29.1.1 u_perror()

```
void u_perror (  {\tt const\ char\ *\ msg\ )}
```

Creates a user-level error message similar to perror.

Parameters

msg A string representing the error message from the shell.

Definition at line 15 of file builtins.c.

```
char buffer[256];
16
17
       const char *error_msg;
18
       switch (P_ERRNO) {
19
20
           case P_ENOENT:
21
               error_msg = "file does not exist";
22
           case P_EBADF:
2.3
               error_msg = "bad file descriptor";
24
25
               break:
           case P_EPERM:
26
27
               error_msg = "operation not permitted";
28
               break;
29
           case P_EINVAL:
               error_msg = "invalid arq";
30
31
               break;
           case P_EEXIST:
32
               error_msg = "file already exists";
34
35
           case P_EBUSY:
              error_msg = "file is busy or open";
36
37
               break;
38
           case P_EFULL:
              error_msg = "no space left on device";
40
41
           case P_EINTR:
               error_msg = "interrupted system call";
42
43
               break:
           case P_ENULL:
44
45
               error_msg = "NULL returned unexpectedly";
46
47
           case P_EUNKNOWN:
               error_msg = "unknown error";
48
49
               break;
           case P_EREAD:
50
               error_msg = "interrupted read call";
               break;
           case P_ELSEEK:
               error_msg = "interrupted lseek call";
54
5.5
               break;
           case P EMAP:
56
               error_msg = "interrupted mmap/munmap call";
               break;
59
           case P_EFUNC:
60
              error_msg = "interrupted system call";
               break;
61
           case P_EOPEN:
62
              error_msg = "interrupted open call";
63
65
           case P_EMALLOC:
66
               error_msg = "error when trying to malloc";
67
           break;
case P_EFS_NOT_MOUNTED:
68
               error_msg = "file system not mounted yet";
69
               break;
           case P_ESIGNAL:
71
               error_msg = "error with signal handling";
72
               break;
73
           case P EWRITE:
74
               error_msg = "interrupted write call";
75
               break;
           case P_ECLOSE:
               error_msg = "interrupted close call";
78
79
               break;
           case P_EPARSE:
80
               error_msg = "error when trying to parse a command";
81
82
               break:
           case P_ECOMMAND:
              error_msg = "command not found";
84
               break;
85
86
           case P_NEEDF:
              error_msg = "no file provided to mount";
87
88
               break:
           case P_EREDIR:
90
               error_msg = "input and output cannot be the same when appending";
               break;
91
92
           default:
               error_msg = "Unknown error";
93
94
               break;
95
       }
       snprintf(buffer, sizeof(buffer), "%s: %s\n", msg, error_msg);
97
98
       if (s_write(STDERR_FILENO, buffer, strlen(buffer)) == -1) {
99
           perror("s_write");
        }
100
```

101 }

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)
 Creates a user-level error message similar to perror.

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 15 of file builtins.c.

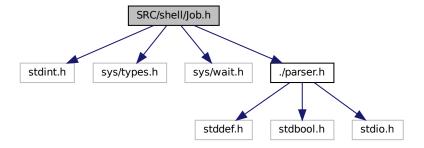
```
16
       char buffer[256];
17
      const char *error msg;
18
19
     switch (P_ERRNO) {
      case P_ENOENT:
20
              error_msg = "file does not exist";
21
22
          case P_EBADF:
23
              error_msg = "bad file descriptor";
26
27
           case P_EPERM:
               error_msg = "operation not permitted";
          break;
case P_EINVAL:
28
29
             error_msg = "invalid arg";
              break;
           case P_EEXIST:
              error_msg = "file already exists";
33
           break;
case P_EBUSY:
34
35
              error_msg = "file is busy or open";
36
               break;
```

```
case P_EFULL:
              error_msg = "no space left on device";
40
                break;
           case P_EINTR:
41
              error_msg = "interrupted system call";
42
43
                break:
           case P_ENULL:
45
                error_msg = "NULL returned unexpectedly";
46
47
           case P_EUNKNOWN:
                error_msg = "unknown error";
48
49
               break:
           case P_EREAD:
50
              error_msg = "interrupted read call";
                break;
53
           case P_ELSEEK:
                error_msg = "interrupted lseek call";
54
55
               break;
           case P_EMAP:
56
                error_msg = "interrupted mmap/munmap call";
59
           case P_EFUNC:
              error_msg = "interrupted system call";
60
61
               break;
           case P_EOPEN:
              error_msg = "interrupted open call";
63
65
           case P_EMALLOC:
                error_msg = "error when trying to malloc";
66
67
                break:
           case P_EFS_NOT_MOUNTED:
68
69
                error_msg = "file system not mounted yet";
70
71
           case P_ESIGNAL:
                error_msg = "error with signal handling";
72
73
                break;
           case P_EWRITE:
74
75
                error_msg = "interrupted write call";
                break;
77
           case P_ECLOSE:
                error_msg = "interrupted close call";
78
           break;
case P_EPARSE:
79
80
              error_msg = "error when trying to parse a command";
               break;
83
           case P_ECOMMAND:
              error_msg = "command not found";
84
           break;
case P_NEEDF:
8.5
86
              error_msg = "no file provided to mount";
87
89
           case P_EREDIR:
               error_msg = "input and output cannot be the same when appending";
90
91
                break;
92
           default:
                error_msg = "Unknown error";
93
                break;
95
96
       snprintf(buffer, sizeof(buffer), "%s: %s\n", msg, error_msg); \\ if (s_write(STDERR_FILENO, buffer, strlen(buffer)) == -1) \{
97
98
           perror("s_write");
99
100
101 }
```

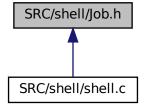
4.31 SRC/shell/Job.h File Reference

```
#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	

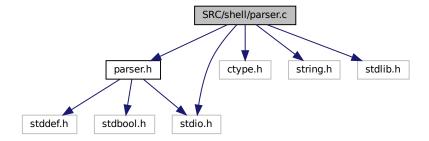
```
Definition at line 13 of file Job.h.
```

```
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 }
```

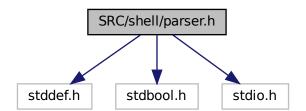
4.32.2.3 print_parser_errcode()

```
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 }
```

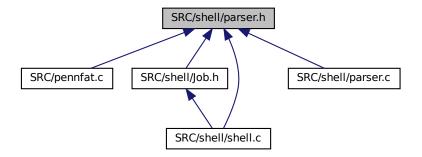
4.33 SRC/shell/parser.h File Reference

```
#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()

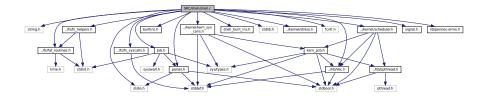
```
void print_parser_errcode (
               FILE * output,
               int err_code )
Definition at line 230 of file parser.c.
230
      switch (err_code) {
       case UNEXPECTED_FILE_INPUT:
233
          fprintf(output, "UNEXPECTED INPUT REDIRECTION TO A FILE\n");
234
        case UNEXPECTED_FILE_OUTPUT:
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
- int is_append = 0

4.34.1 Macro Definition Documentation

4.34.1.1 MAX_BUFFER_SIZE

```
#define MAX_BUFFER_SIZE 4096
```

Definition at line 30 of file shell.c.

4.34.1.2 MAX_LINE_BUFFER_SIZE

```
#define MAX_LINE_BUFFER_SIZE 128
```

Definition at line 31 of file shell.c.

4.34.1.3 PROMPT

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

Definition at line 27 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 271 of file shell.c.

```
271
272
273  // setup fds
274  int input_fd = STDIN_FILENO; // standard fds
275  int output_fd = STDOUT_FILENO;
276
```

```
if (cmd->stdin_file != NULL) {
278
         input_fd = s_open(cmd->stdin_file, F_READ);
             (input_fd < 0) {
279
           input_fd = STDIN_FILENO; // reset to default
280
281
282
      }
283
284
       if (cmd->is_file_append) {
         output_fd = s_open(cmd->stdout_file, F_APPEND);
is_append = 1;
285
286
287
       } else {
         output_fd = s_open(cmd->stdout_file, F_WRITE);
288
289
         is_append = 0;
290
291
       if (output_fd < 0) {</pre>
292
         output_fd = STDOUT_FILENO; // reset to default
293
294
295
       // check for independently scheduled processes
296
       if (strcmp(cmd->commands[0][0], "cat") == 0) {
297
         return s_spawn(u_cat, cmd->commands[0], input_fd, output_fd);
298
       } else if (strcmp(cmd->commands[0][0], "sleep") == 0) {
         return s_spawn(u_sleep, cmd->commands[0], input_fd, output_fd);
299
       } else if (strcmp(cmd->commands[0], "busy") == 0) {
  return s_spawn(u_busy, cmd->commands[0], input_fd, output_fd);
300
301
       } else if (strcmp(cmd->commands[0][0], "echo") == 0) {
303
         return s_spawn(u_echo, cmd->commands[0], input_fd, output_fd);
       } else if (strcmp(cmd->commands[0][0], "ls") == 0) {
304
       return s_spawn(u_ls, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "touch") == 0) {
305
306
         return s_spawn(u_touch, cmd->commands[0], input_fd, output_fd);
307
308
       } else if (strcmp(cmd->commands[0][0], "mv") == 0) {
       return s_spawn(u_mv, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "cp") == 0) {
309
310
       return s_spawn(u_cp, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "rm") == 0) {
311
312
       return s_spawn(u_rm, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "chmod") == 0) {
313
314
315
         return s_spawn(u_chmod, cmd->commands[0], input_fd, output_fd);
       } else if (strcmp(cmd->commands[0][0], "ps") == 0) {
316
       return s_spawn(u_ps, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "kill") == 0) {
317
318
       return s_spawn(u_kill, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "zombify") == 0) {
319
320
         return s_spawn(u_zombify, cmd->commands[0], input_fd, output_fd);
322
       } else if (strcmp(cmd->commands[0][0], "orphanify") == 0)
323
         return s_spawn(u_orphanify, cmd->commands[0], input_fd, output_fd);
324
       } else if (strcmp(cmd->commands[0][0], "hang") == 0) {
       return s_spawn(hang, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "nohang") == 0) {
   return s_spawn(nohang, cmd->commands[0], input_fd, output_fd);
}
325
326
327
328
       } else if (strcmp(cmd->commands[0][0], "recur") == 0) {
329
         return s_spawn(recur, cmd->commands[0], input_fd, output_fd);
       } else if (strcmp(cmd->commands[0][0], "crash") == 0) {
  return s_spawn(crash, cmd->commands[0], input_fd, output_fd);
330
331
332
333
334
       // check for sub-routines
335
       if (strcmp(cmd->commands[0][0], "nice") == 0) {
         u_nice(cmd->commands[0]);
336
337
         return 0:
       } else if (strcmp(cmd->commands[0][0], "nice_pid") == 0) {
338
339
         u_nice_pid(cmd->commands[0]);
340
         return 0;
341
       } else if (strcmp(cmd->commands[0][0], "man") == 0) {
342
         u_man(cmd->commands[0]);
343
         return 0;
       } else if (strcmp(cmd->commands[0][0], "bq") == 0) {
344
345
         u_bg(cmd->commands[0]);
346
         return 0;
347
       } else if (strcmp(cmd->commands[0][0], "fg") == 0) {
348
         u_fg(cmd->commands[0]);
       return 0;
} else if (strcmp(cmd->commands[0][0], "jobs") == 0) {
349
350
351
         u jobs(cmd->commands[0]);
352
         return 0;
353
       } else if (strcmp(cmd->commands[0][0], "logout") == 0) {
354
         u_logout (cmd->commands[0]);
355
         return 0:
356
357
358
       // otherwise, try to run command as a script
       int script_fd_open = s_open(cmd->commands[0][0], F_READ);
359
       if (script_fd_open < 0) { // if not a file, just move on</pre>
360
361
         return -1;
362
       if (has executable permission(script fd open) != 1) {
363
```

```
364
        if (s_close(script_fd_open) == -1) {
365
          u_perror("s_close error i.e. not a valid fd");
366
367
        return -1;
368
      } else {
        script_fd = script_fd_open; // update global
369
370
        input_fd_script = input_fd;
371
        output_fd_script = output_fd;
372
        char* script_argv[] = {cmd->commands[0][0], NULL};
373
374
        pid_t wait_on = s_spawn(u_read_and_execute_script, script_argv, input_fd,
375
                 output_fd);
376
        int status;
        swaitpid(wait_on, &status, false); // wait for script to finish
script_fd = -1; // reset global
377
378
        input_fd_script = STDIN_FILENO;
output_fd_script = STDOUT_FILENO;
379
380
381
        if (s_close(script_fd_open) == -1) {
          u_perror("s_close error i.e. not a valid fd");
382
383
384
385
386
387
      return -1; // no matches case
388 }
```

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 127 of file shell.c.

```
127
128
       int i = 0;
129
       char currChar;
       while (i < MAX_LINE_BUFFER_SIZE - 1) {</pre>
130
        int bytes_read = s_read(fd, &currChar, 1);
if (bytes_read <= 0 || currChar == '\n') { // EOF or newline cases</pre>
131
132
133
           break;
134
135
         buffer[i] = currChar;
136
        i++;
137
      buffer[i] = ' \setminus 0'; // Null-terminate the string, replaces \setminus n
138
```

4.34.2.3 free_job_ptr()

```
void free_job_ptr (
     void * ptr )
```

Definition at line 108 of file shell.c. 108

```
109     job* job_ptr = (job*)ptr;
110     free(job_ptr->pids);
111     free(job_ptr);
112 }
```

4.34.2.4 setup_terminal_signal_handlers()

```
80
      struct sigaction sa_int = {0};
      sa_int.sa_handler = shell_sigint_handler;
81
      sigemptyset(&sa_int.sa_mask);
82
     sa_int.sa_flags = SA_RESTART;
if (sigaction(SIGINT, &sa_int, NULL) == -1) {
83
84
        P_ERRNO = P_ESIGNAL;
86
        u_perror("sigaction");
87
        exit(EXIT_FAILURE);
    }
88
89
    struct sigaction sa_stp = {0};
sa_stp.sa_handler = shell_sigstp_handler;
90
     sigemptyset(&sa_stp.sa_mask);
    sa_stp.sa_flags = SA_RESTART;
94
     if (sigaction(SIGTSTP, &sa_stp, NULL) == -1) {
      P_ERRNO = P_ESIGNAL;
u_perror("sigaction");
95
96
       exit(EXIT_FAILURE);
99 }
```

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 395 of file shell.c.

```
396
397
       job_list = vec_new(0, free_job_ptr);
398
399
      setup_terminal_signal_handlers();
400
401
      while (true) {
402
        int status;
403
         pid_t child_pid;
404
         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
405
406
        }
407
408
409
410
         if (s_write(STDOUT_FILENO, PROMPT, strlen(PROMPT)) < 0) {</pre>
```

```
411
             u_perror("prompt s_write error");
412
413
414
         // parse user input
415
         char buffer[MAX_BUFFER_SIZE];
416
         ssize_t user_input = s_read(STDIN_FILENO, buffer, MAX_BUFFER_SIZE);
417
418
         if (user_input < 0) {</pre>
419
          u_perror("shell read error");
420
           break;
        } else if (user_input == 0) { // EOF case
421
          s_shutdown_pennos();
422
423
          break;
424
425
        buffer[user_input] = '\0';
if (buffer[user_input - 1] == '\n') {
  buffer[user_input - 1] = '\0';
426
427
428
429
430
431
         struct parsed_command* cmd = NULL;
         int cmd_parse_res = parse_command(buffer, &cmd);
if (cmd_parse_res != 0 || cmd == NULL) {
432
433
          P ERRNO = P EPARSE:
434
           u_perror("parse_command");
435
436
           continue;
437
438
439
         // handle the command
440
         if (cmd->num_commands == 0) {
441
          free (cmd):
442
           continue;
443
444
         child_pid = execute_command(cmd);
if (child_pid < 0) {</pre>
445
446
447
          free(cmd);
448
           continue;
449
         } else if (child_pid == 0) {
450
          free(cmd);
451
           continue;
        }
452
453
454
         // If background, add the process to the job list.
         if (cmd->is_background) {
455
456
           // Create a new job entry.
           job* new_job = malloc(sizeof(job));
if (new_job == NULL) {
  perror("Error: mallocing new_job failed");
457
458
459
460
             free (cmd);
461
             continue;
462
463
           new_job->id = next_job_id++;
           new_job->pgid = child_pid; // For single commands, child's pid = pgid.
464
           new_job->num_pids = 1;
465
           new_job->pids = malloc(sizeof(pid_t));
466
           if (new_job->pids == NULL) {
467
468
             perror("Error: mallocing new_job->pids failed");
469
              free(new_job);
470
             free (cmd);
471
             continue;
472
473
           new_job->pids[0] = child_pid;
           new_job->state = RUNNING;
new_job->cmd = cmd; // Retain command info; do not free here.
474
475
           new_job->finished_count = 0;
476
477
           vec_push_back(&job_list, new_job);
478
479
           // Print job control information in the format: "[job_id] child_pid"
           char msg[128];
480
481
           snprintf(msg, sizeof(msg), "[%lu] %d\n", (unsigned long)new_job->id,
                     child_pid);
482
           if (s_write(STDOUT_FILENO, msg, strlen(msg)) == -1) {
  u_perror("s_write error");
483
484
485
486
         } else {
487
           // Foreground execution.
488
           current_fg_pid = child_pid;
489
           int status;
490
           s waitpid (child pid, &status, false);
491
           current_fg_pid = 2;
492
        }
493
494
495
      vec_destroy(&job_list);
      s_exit();
496
497
      return 0:
```

498 }

63 }

4.34.2.6 shell sigint handler()

4.34.2.7 shell_sigstp_handler()

```
void shell_sigstp_handler ( \quad \text{ int } sig \ )
```

Definition at line 67 of file shell.c.

```
// If there's a foreground process, forward SIGTSTP (stop) to it
if (current_fg_pid != 2) {
    s_kill(current_fg_pid, 0); // P_SIGSTOP
}

if (s_write(STDOUT_FILENO, "\n", 1) == -1) {
    u_perror("s_write error");
}
```

4.34.2.8 u_execute_command()

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 150 of file shell.c.
151
        // 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) {
152
153
154
155
           return s_spawn(u_sleep, cmd->commands[0], input_fd_script, output_fd_script);
156
157
        } 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) {
158
159
        return s_spawn(u_echo, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "ls") == 0) {
160
161
        return s_spawn(u_ls, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "touch") == 0) {
163
        return s_spawn(u_touch, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "mv") == 0) {
164
165
        return s_spawn(u_mv, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "cp") == 0) {
166
167
        return s_spawn(u_cp, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "rm") == 0) {
168
169
        return s_spawn(u_rm, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "chmod") == 0) {
170
171
        return s_spawn(u_chmod, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "ps") == 0) {
   return s_spawn(u_ps, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "kill") == 0) {
172
173
174
175
176
           return s_spawn(u_kill, cmd->commands[0], input_fd_script, output_fd_script);
        } else if (strcmp(cmd->commands[0][0], "zombify") == 0) {
177
178
           return s_spawn(u_zombify, cmd->commands[0], input_fd_script, output_fd_script);
        less if (strcmp(cmd->commands[0][0], "orphanify") == 0) {
  return s_spawn(u_orphanify, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "hang") == 0) {
179
180
181
        return s_spawn(hang, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "nohang") == 0) {
182
183
        return s_spawn(nohang, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "recur") == 0) {
184
185
          return s_spawn(recur, cmd->commands[0], input_fd_script, output_fd_script);
186
187
        } else if (strcmp(cmd->commands[0][0], "crash") == 0) {
188
          return s_spawn(crash, cmd->commands[0], input_fd_script, output_fd_script);
189
190
        // check for sub-routines
191
        if (strcmp(cmd->commands[0][0], "nice") == 0) {
192
193
          u_nice(cmd->commands[0]);
194
           return 0;
195
        } else if (strcmp(cmd->commands[0][0], "nice_pid") == 0) {
196
          u_nice_pid(cmd->commands[0]);
197
           return 0:
        } else if (strcmp(cmd->commands[0][0], "man") == 0) {
198
199
           u_man(cmd->commands[0]);
200
           return 0;
201
        } else if (strcmp(cmd->commands[0][0], "bg") == 0) {
202
           u_bg(cmd->commands[0]);
203
           return 0;
        } else if (strcmp(cmd->commands[0][0], "fg") == 0) {
204
205
          u_fg(cmd->commands[0]);
206
           return 0;
207
        } else if (strcmp(cmd->commands[0][0], "jobs") == 0) {
208
           u_jobs(cmd->commands[0]);
209
           return 0;
        } else if (strcmp(cmd->commands[0][0], "logout") == 0) {
210
211
          u_logout (cmd->commands[0]);
212
           return 0;
213
        } else {
214
           return -1; // no matches, no scripts now
```

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.

215 216 217

218 }

return 0:

Parameters

arg standard (function name, NULL) args

Returns

NULL

```
Definition at line 228 of file shell.c.
229
       // read the script line by line, parse each line, and execute the command
230
       while (true) {
        char buffer[MAX_LINE_BUFFER_SIZE];
231
        fill_buffer_until_full_or_newline(script_fd, buffer);
if (buffer[0] == '\0') {
232
233
234
           break; // EOF case
235
236
237
        // parse the command
        struct parsed_command* cmd = NULL;
238
        int parse_result = parse_command(buffer, &cmd);
if (parse_result != 0 || cmd == NULL) {
239
240
         P_ERRNO = P_EPARSE;
u_perror("parse_command");
free(cmd);
241
242
243
2.44
245
        // execute the command
246
247
        pid_t child_pid = u_execute_command(cmd);
248
         if (child_pid > 0) { // if process was spawned, wait for it to finish
249
         int status;
       s_waitpid(child_pid, &status, false);
} else if (child_pid < 0) { // nothing spawning so safe to free cmd</pre>
250
251
252
           free (cmd);
253
      }
254
255
      s_exit(); // exit the script
256
257
      return NULL;
258 }
```

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 43 of file shell.c.

4.34.3.3 is_append

```
int is_append = 0
```

Definition at line 45 of file shell.c.

4.34.3.4 job_list

```
Vec job_list
```

Definition at line 38 of file shell.c.

4.34.3.5 next_job_id

```
jid_t next_job_id = 1
```

Definition at line 39 of file shell.c.

4.34.3.6 output_fd_script

```
int output_fd_script = -1
```

Definition at line 44 of file shell.c.

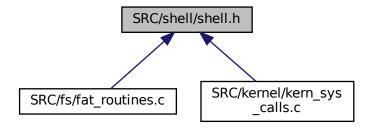
4.34.3.7 script_fd

```
int script_fd = -1
```

Definition at line 42 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.

Variables

· int is_append

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 395 of file shell.c.

```
396
397
       job_list = vec_new(0, free_job_ptr);
398
399
       setup_terminal_signal_handlers();
401
       while (true) {
402
         int status;
403
         pid_t child_pid;
         while ((child_pid = s_waitpid(-1, &status, true)) > 0) {
   // Child process has completed, no need to do anything special
404
405
406
           // The s_waitpid function already handles cleanup
407
408
         // prompt
409
410
         if (s_write(STDOUT_FILENO, PROMPT, strlen(PROMPT)) < 0) {</pre>
              u_perror("prompt s_write error");
411
              break;
413
414
         // parse user input
char buffer[MAX_BUFFER_SIZE];
415
416
         ssize_t user_input = s_read(STDIN_FILENO, buffer, MAX_BUFFER_SIZE);
417
418
         if (user_input < 0) {</pre>
          u_perror("shell read error");
         break;
} else if (user_input == 0) { // EOF case
420
421
422
           s_shutdown_pennos();
423
           break:
424
425
         buffer[user_input] = '\0';
if (buffer[user_input - 1] == '\n') {
  buffer[user_input - 1] = '\0';
426
427
428
429
430
```

```
431
        struct parsed_command* cmd = NULL;
        int cmd_parse_res = parse_command(buffer, &cmd);
if (cmd_parse_res != 0 || cmd == NULL) {
432
433
         P_ERRNO = P_EPARSE;
u_perror("parse_command");
434
435
436
          continue:
437
438
439
        // handle the command
440
        if (cmd->num_commands == 0) {
         free(cmd);
441
442
          continue:
443
444
445
        child_pid = execute_command(cmd);
        if (child_pid < 0) {</pre>
446
447
          free (cmd);
448
        continue;
} else if (child_pid == 0) {
449
450
          free(cmd);
451
          continue;
452
453
        \ensuremath{//} If background, add the process to the job list.
454
455
        if (cmd->is_background) {
          // Create a new job entry.
456
457
           job* new_job = malloc(sizeof(job));
458
           if (new_job == NULL) {
459
             perror("Error: mallocing new_job failed");
460
             free (cmd);
461
             continue:
462
463
           new_job->id = next_job_id++;
464
           new_job->pgid = child_pid; // For single commands, child's pid = pgid.
465
           new_job->num_pids = 1;
           new_job->pids = malloc(sizeof(pid_t));
if (new_job->pids == NULL) {
466
467
            perror("Error: mallocing new_job->pids failed");
468
469
             free(new_job);
470
            free(cmd);
471
             continue;
472
473
          new_job->pids[0] = child_pid;
          new_job->state = RUNNING;
new_job->cmd = cmd; // Retain command info; do not free here.
474
475
476
           new_job->finished_count = 0;
477
           vec_push_back(&job_list, new_job);
478
479
           // Print job control information in the format: "[job_id] child_pid"
           char msg[128];
480
           snprintf(msg, sizeof(msg), "[%lu] %d\n", (unsigned long)new_job->id,
481
482
                    child_pid);
483
           if (s\_write(STDOUT\_FILENO, msg, strlen(msg)) == -1) {
484
            u_perror("s_write error");
485
486
        } else {
          // Foreground execution.
487
488
           current_fg_pid = child_pid;
489
           int status;
490
           s_waitpid(child_pid, &status, false);
491
          current_fg_pid = 2;
492
493
494
495
      vec_destroy(&job_list);
496
      s_exit();
497
      return 0;
498 }
```

4.35.2 Variable Documentation

4.35.2.1 is append

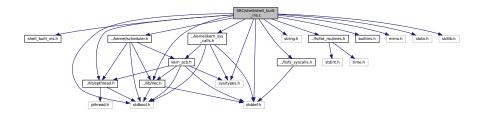
int is_append [extern]

Definition at line 45 of file shell.c.

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 "builtins.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 321 of file shell_built_ins.c.

```
321
322 while (1)
323 ;
324 s_exit();
325 }
```

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 278 of file shell built ins.c.

```
278 {
279 // TODO --> implement bg
280 return NULL;
281 }
```

4.36.1.3 u_busy()

Busy wait indefinitely. It can only be interrupted via signals.

Example Usage: busy

Definition at line 54 of file shell_built_ins.c.

```
55 while (1)

56 ;

57 s_exit();

58 return NULL;

59 }
```

4.36.1.4 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 29 of file shell_built_ins.c.

```
29
30    cat(arg);
31    s_exit();
32    return NULL;
33 }
```

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 73 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 91 of file shell_built_ins.c.

```
91 {
92 cp(arg);
93 s_exit();
94 return NULL;
95 }
```

4.36.1.7 u_echo()

```
void* u_echo (
          void * arg )
```

Echo back an input string.

Example Usage: echo Hello World

Definition at line 61 of file shell_built_ins.c.

```
61 {
62    s_echo(arg);
63    s_exit();
64    return NULL;
65 }
```

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 283 of file shell_built_ins.c.

```
283 {
284  // TODO --> implement fg
285  return NULL;
286 }
```

4.36.1.9 u_jobs()

```
void* u_jobs (
     void * arg )
```

Lists all jobs.

Example Usage: jobs

Definition at line 288 of file shell_built_ins.c.

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 109 of file shell_built_ins.c.

```
110
        char** argv = (char**)arg;
       int sig = 2;
111
112
       char err_buf[128];
113
114
115
       // Check if the first argument specifies a signal
116
       if (argv[start_index] && argv[start_index][0] == '-') {
   if (strcmp(argv[start_index], "-term") == 0) {
117
         sig = 2;
} else if (strcmp(argv[start_index], "-stop") == 0) {
118
119
120
            sig = 0;
121
         } else if (strcmp(argv[start_index], "-cont") == 0) {
122
            sig = 1;
         } else {
   // Construct error message
123
124
125
            s exit();
126
            return NULL;
127
128
          start_index++;
129
130
       // Process each PID argument using strtol
131
132
       for (int i = start_index; argv[i] != NULL; i++) {
133
         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>
134
135
136
            if (s_write(STDERR_FILENO, err_buf, strlen(err_buf)) == -1) {
   u_perror("s_write error");
137
138
139
140
            continue;
141
142
          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);
    if (s_write(STDERR_FILENO, err_buf, strlen(err_buf)) == -1) {
        u_perror("s_write error");
    }
}</pre>
143
144
145
146
147
      }
148
149
       s exit();
150
       return NULL;
151
```

4.36.1.11 u_logout()

Exits the shell and shutsdown PennOS.

Example Usage: logout

Definition at line 293 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 67 of file shell built ins.c.

```
67 {
68 ls(arg);
69 s_exit();
70 return NULL;
71 }
```

4.36.1.13 u_man()

Lists all available commands.

Example Usage: man

Definition at line 238 of file shell_built_ins.c.

```
238
      const char* man_string =
239
           "cat f1 f2 ...
240
                                   : concatenates provided files (if none, reads from "
241
          "std in), and writes to std out\n"
                         : sleeps for n seconds\n"
242
           "sleep n
243
           "busy
                                    : busy waits indefinitely\n"
           "echo str
2.44
                                   : echoes back the input string \mathsf{str} \setminus \mathsf{n}"
           "ls : lists all files in the working directory\n"
"touch f1 f2 ... : for each file, creates empty file if it doesn't "
245
246
           "exist yet, otherwise updates its timestamp\n"
247
                          : renames f1 to f2 (overwrites f2 if it exists)\n"
: copies f1 to f2 (overwrites f2 if it exists)\n"
248
           "mv f1 f2
249
           "cp f1 f2
           "rm f1 f2 ...
"chmod +_ f1
250
                                   : removes the input list of files\n"
251
                                   : changes fl permissions to +_ specifications "
252
           "(+x, +rw, etc)\n"
253
           "ps
                                   : lists all processes on PennOS, displaying PID, "
254
          "PPID, priority, status, and command name\n"
255
           "kill (-__) pid1 pid 2 : sends specified signal (term default) to list "
256
           "of processes\n"
           "nice n command
2.57
                                    : spawns a new process for command and sets its "
258
           "priority to n\n'
259
           "nice_pid n pid
                                   : adjusts the priority level of an existing
260
           "process to n \ n"
261
                                    : lists all available commands in PennOS\n"
           "man
                                    : resumes most recently stopped process in "
262
           "bg
           "background or the one specified by job_id\n"
"fg : brings most recently stopped or background job"
263
264
           "to foreground or the one specifed by job_id\n"
265
266
           "jobs
                                   : lists all jobs\n"
267
           "logout
                                    : exits the shell and shuts down PennOS\n"
268
           "zombify
                                    : creates a child process that becomes a zombie \n"
           "orphanify
269
                                   : creates a child process that becomes an
270
           "orphan\n";
271
272
      if (s_write(STDOUT_FILENO, man_string, strlen(man_string)) == -1) {
        u_perror("s_write error");
273
274
275
      return NULL;
276 }
```

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 85 of file shell_built_ins.c.

4.36.1.15 u_nice()

```
void* u_nice (
     void * arg )
```

Spawn a new process for command and set its priority to priority.

Example Usage: nice 2 cat f1 f2 f3 (spawns cat with priority 2)

Definition at line 199 of file shell built ins.c.

```
199
200
      char* endptr;
201
       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>
202
203
204
205
       return NULL;
206
207
208
      char* command = ((char**)arg)[2];
      void* (*ufunc)(void*) = get_associated_ufunc(command);
209
210
      if (ufunc == NULL) {
        return NULL; // no matches, don't spawn
211
212
213
214
      pid_t new_proc_pid = s_spawn(ufunc, &((char**)arg)[2], 0, 1);
215
216
      if (new_proc_pid != -1) { // non-error case
     217
218
219
220
      return NULL;
221 }
```

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 223 of file shell_built_ins.c.

```
223
224
         char* endptr;
225
         errno = 0;
        int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10); if (*endptr != '\0' || errno != 0) { // error catch
226
227
          return NULL;
228
229
        pid_t pid = (pid_t)strtol(((char**)arg)[2], &endptr, 10);
if (*endptr != '\0' || errno != 0) {
   return NULL;
230
231
232
233
234 s_nice(pid, new_priority);
235 return NULL;
236 }
```

4.36.1.17 u_orphanify()

Used to test orphanifying functionality of your kernel.

Example Usage: orphanify

Definition at line 327 of file shell_built_ins.c.

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 103 of file shell_built_ins.c.

```
103 {
104    s_ps(arg);
105    s_exit();
106    return NULL;
107 }
```

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 97 of file shell_built_ins.c.

```
97 {
98 rm(arg);
99 s_exit();
100 return NULL;
101 }
```

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 35 of file shell built ins.c.

```
36
      char* endptr;
     errno = 0;
if (((char**)arg)[1] == NULL) { // no arg case
37
38
39
       s_exit();
40
        return NULL;
    int sleep_secs = (int)strtol(((char**)arg)[1], &endptr, 10);
if (*endptr != '\0' || errno != 0 || sleep_secs <= 0) {
43
44
        s_exit();
        return NULL;
4.5
46
     int sleep_ticks = sleep_secs * 10;
     s_sleep(sleep_ticks);
49
50
    s_exit();
     return NULL;
51
52 }
```

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 79 of file shell_built_ins.c.

```
79 {
80 touch(arg);
81 s_exit();
82 return NULL;
83 }
```

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 310 of file shell_built_ins.c.

4.36.1.23 zombie_child()

Helper for zombify.

Definition at line 305 of file shell built ins.c.

```
305 {
306 s_exit();
307 return NULL;
308 }
```

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

190

191

193

194 195

196 197 }

192

A ptr to the associated u-version function or NULL if no matches are found

Definition at line 169 of file shell_built_ins.c. 170 if (strcmp(func, "cat") == 0) { return u_cat;
} else if (strcmp(func, "sleep") == 0) { 171 172 173 return u sleep; 174 } else if (strcmp(func, "busy") == 0) { 175 return u_busy; 176 } else if (strcmp(func, "echo") == 0) { 177 return u_echo; 178 } else if (strcmp(func, "ls") == 0) { 179 return u_ls; } else if (strcmp(func, "touch") == 0) { return u_touch; } else if (strcmp(func, "mv") == 0) { return u_mv;
} else if (strcmp(func, "cp") == 0) { 183 184 185 return u cp; } else if (strcmp(func, "rm") == 0) { 186 return u_rm; 188 } else if (strcmp(func, "chmod") == 0) { 189 return u_chmod;

} else if (strcmp(func, "ps") == 0) {

return NULL; // no matches case

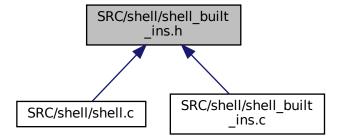
} else if (strcmp(func, "kill") == 0) {

return u_ps;

return u_kill;

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)
```

4.37.1 Function Documentation

void * u_orphanify (void *arg)

Used to test zombifying functionality of your kernel.

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 278 of file shell built ins.c.

```
279  // TODO --> implement bg
280  return NULL;
281 }
```

4.37.1.2 u_busy()

Busy wait indefinitely. It can only be interrupted via signals.

Example Usage: busy

Definition at line 54 of file shell_built_ins.c.

```
54 {
55 while (1)
56 ;
57 s_exit();
58 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 29 of file shell_built_ins.c.

```
29
30    cat(arg);
31    s_exit();
32    return NULL;
33 }
```

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 73 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 91 of file shell_built_ins.c.

```
91 {
92 cp(arg);
93 s_exit();
94 return NULL;
95 }
```

4.37.1.6 u_echo()

```
void* u_echo (
          void * arg )
```

Echo back an input string.

Example Usage: echo Hello World

Definition at line 61 of file shell_built_ins.c.

```
61 {
62    s_echo(arg);
63    s_exit();
64    return NULL;
65 }
```

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 283 of file shell_built_ins.c.

```
283 {
284  // TODO --> implement fg
285  return NULL;
286 }
```

4.37.1.8 u_jobs()

```
void* u_jobs (
     void * arg )
```

Lists all jobs.

Example Usage: jobs

Definition at line 288 of file shell_built_ins.c.

```
288 {
289    // TODO --> implement jobs
290    return NULL;
291 }
```

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 109 of file shell_built_ins.c.

```
110
        char** argv = (char**)arg;
        int sig = 2; // Default signal: term (2) int start_index = 1; // Start after the "kill" command word.
111
112
        char err_buf[128];
113
114
115
        // Check if the first argument specifies a signal
116
        if (argv[start_index] && argv[start_index][0] == '-') {
   if (strcmp(argv[start_index], "-term") == 0) {
117
          sig = 2;
} else if (strcmp(argv[start_index], "-stop") == 0) {
118
119
120
             sig = 0;
121
          } else if (strcmp(argv[start_index], "-cont") == 0) {
122
             sig = 1;
          } else {
   // Construct error message
123
124
125
             s exit();
126
             return NULL;
127
128
           start_index++;
129
130
        // Process each PID argument using strtol
131
        for (int i = start_index; argv[i] != NULL; i++) {
132
133
          char* endptr;
           ind 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
             if (s_write(STDERR_FILENO, err_buf, strlen(err_buf)) == -1) {
    u_perror("s_write error");
137
138
139
140
             continue;
141
142
           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);
    if (s_write(STDERR_FILENO, err_buf, strlen(err_buf)) == -1) {
        u_perror("s_write error");
    }
}</pre>
143
144
145
146
147
       }
148
149
       s exit();
150
151
        return NULL;
```

4.37.1.10 u_logout()

Exits the shell and shutsdown PennOS.

Example Usage: logout

Definition at line 293 of file shell_built_ins.c.

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 67 of file shell built ins.c.

```
67 {
68 ls(arg);
69 s_exit();
70 return NULL;
71 }
```

4.37.1.12 u_man()

Lists all available commands.

Example Usage: man

Definition at line 238 of file shell_built_ins.c.

```
238
       const char* man_string =
239
240
            "cat f1 f2 ...
                                    : concatenates provided files (if none, reads from "
241
           "std in), and writes to std out\n"
                         : sleeps for n seconds \n"
242
           "sleep n
243
           "busy
                                     : busy waits indefinitely\n"
2.44
           "echo str
                                     : echoes back the input string \mathsf{str} \setminus \mathsf{n}"
           "ls : lists all files in the working directory\n"
"touch f1 f2 ... : for each file, creates empty file if it doesn't "
245
246
           "mv f1 f2 : renames f1 to f2 (overwrites f2 if it exists)\n"
"rm f1 f2 : copies f1 to f2 (overwrites f2 if it exists)\n"
"rm f1 f2 : removes the input list of files\n"
"chmod +_ f1 : changes f1 ---
247
248
249
250
251
           "(+x, +rw, etc)\n"
252
           "ps
                                    : lists all processes on PennOS, displaying PID, "
254
           "PPID, priority, status, and command name\n"
255
           "kill (-__) pid1 pid 2 : sends specified signal (term default) to list "
256
           "of processes\n"
           "nice n command
2.57
                                     : spawns a new process for command and sets its "
258
            "priority to n\n"
259
           "nice_pid n pid
                                     : adjusts the priority level of an existing
260
           "process to n \ n"
261
                                     : lists all available commands in PennOS\n"
           "man
                                     : resumes most recently stopped process in "
262
           "bg
            "background or the one specified by job_id\n"
"fg : brings most recently stopped or background job"
263
264
           "to foreground or the one specifed by job_id\n"
265
266
           "jobs
                      : lists all jobs\n"
267
           "logout
                                     : exits the shell and shuts down PennOS\n"
268
           "zombify
                                     : creates a child process that becomes a zombie \n"
           "orphanify
269
                                     : creates a child process that becomes an
270
            "orphan\n";
271
272
      if (s_write(STDOUT_FILENO, man_string, strlen(man_string)) == -1) {
        u_perror("s_write error");
273
274
275
       return NULL;
276 }
```

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 85 of file shell_built_ins.c.

4.37.1.14 u_nice()

```
void* u_nice (
     void * arg )
```

Spawn a new process for command and set its priority to priority.

Example Usage: nice 2 cat f1 f2 f3 (spawns cat with priority 2)

Definition at line 199 of file shell built ins.c.

```
199
200
      char* endptr;
201
       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>
202
203
204
205
206
207
208
      char* command = ((char**)arg)[2];
      void* (*ufunc)(void*) = get_associated_ufunc(command);
209
210
      if (ufunc == NULL) {
        return NULL; // no matches, don't spawn
211
212
213
214
      pid_t new_proc_pid = s_spawn(ufunc, &((char**)arg)[2], 0, 1);
215
216
      if (new_proc_pid != -1) { // non-error case
      217
218
219
220
      return NULL;
221 }
```

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 223 of file shell_built_ins.c.

```
223
224
         char* endptr:
225
         errno = 0;
        int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10); if (*endptr != '\0' || errno != 0) { // error catch
226
227
          return NULL;
228
229
        pid_t pid = (pid_t)strtol(((char**)arg)[2], &endptr, 10);
if (*endptr != '\0' || errno != 0) {
    return NULL;
230
231
232
233
234 s_nice(pid, new_priority);
235 return NULL;
236 }
```

4.37.1.16 u_orphanify()

Used to test orphanifying functionality of your kernel.

Example Usage: orphanify

Definition at line 327 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 103 of file shell_built_ins.c.

{

4.37.1.18 u_rm()

```
void* u_rm ( \label{eq:void*} \mbox{void} * \mbox{\it arg} \mbox{\it )}
```

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 97 of file shell_built_ins.c.

```
97 {
98 rm(arg);
99 s_exit();
100 return NULL;
101 }
```

4.37.1.19 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 35 of file shell built ins.c.

```
36
      char* endptr;
     errno = 0;
if (((char**)arg)[1] == NULL) { // no arg case
37
38
39
       s_exit();
40
        return NULL;
     int sleep_secs = (int)strtol(((char**)arg)[1], &endptr, 10);
if (*endptr != '\0' || errno != 0 || sleep_secs <= 0) {
43
44
        s_exit();
        return NULL;
4.5
46
     int sleep_ticks = sleep_secs * 10;
     s_sleep(sleep_ticks);
49
50
    s_exit();
     return NULL;
51
52 }
```

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 79 of file shell_built_ins.c.

```
touch (arg);
81
82
     s_exit();
return NULL;
83 }
```

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 310 of file shell_built_ins.c.

```
310 {
311 char* zombie_child_argv[] = {"zombie_child", NULL};
312 s_spawn(zombie_child, zombie_child_argv, STDIN_FILENO, STDOUT_FILENO);
313 while (1)
314 ;
315 return NULL;
316 }
```

Index

_GNU_SOURCE	fat_routines.c, 29
spthread.c, 195	fat_routines.h, 46
	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, 62
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
_ ' '	cp
block_size	fat_routines.c, 29
fs_helpers.c, 69	fat routines.h, 47
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, 125
	kern_pcb.h, 130
capacity	curr_priority_arr_index
vec_st, 20	scheduler.c, 170
cat	current_fg_pid
fat_routines.c, 24	fs kfuncs.c, 100
fat_routines.h, 41	kern_sys_calls.c, 143
child_in_zombie_queue	shell.c, 256
scheduler.c, 163	current pcbs
scheduler.h, 174	kern_pcb.c, 128
child_meta	kern_sys_calls.c, 143
spthread_fwd_args_st, 16	scheduler.c, 170
child_pcbs	
pcb_st, 13	current_running_pcb
child_with_changed_process_status	fat_routines.c, 37 fs_kfuncs.c, 100
scheduler.c, 164	-
scheduler.h, 174	kern_pcb.c, 128
chmod	kern_sys_calls.c, 143
fat_routines.c, 28	scheduler.c, 170
fat_routines.h, 45	data
cmd	
job_st, 9	vec_st, 21
cmd_str	decrement_fd_ref_count
pcb_st, 13	fs_helpers.c, 65
cmpctdir	fs_helpers.h, 81 delete from explicit queue
-	ucicle non exonen onene

kern_sys_calls.c, 136	current_running_pcb, 37
kern_sys_calls.h, 146	ls, 30
delete_from_queue	mkfs, 30
kern_sys_calls.c, 136	mount, 32
kern_sys_calls.h, 147	mv, 32
delete_process_from_all_queues	rm, 34
scheduler.c, 164	touch, 35
scheduler.h, 175	unmount, 36
delete_process_from_all_queues_except_current	fat_routines.h
scheduler.c, 164	cat, 41
scheduler.h, 175	chmod, 45
delete_process_from_particular_queue	cmpctdir, 46
scheduler.c, 165	cp, 47
scheduler.h, 176	F APPEND, 38
det_priorities_arr	F READ, 39
scheduler.c, 171	F WRITE, 39
determine_index_in_queue	FAT_EOF, 39
kern_sys_calls.c, 136	FAT FREE, 39
kern_sys_calls.h, 147	ls, 48
dir_entry_t, 5	mkfs, 49
firstBlock, 5	mount, 50
mtime, 5	mv, 51
name, 6	PERM EXEC, 39
perm, 6	PERM NONE, 39
reserved, 6	PERM READ, 40
size, 6	PERM READ EXEC, 40
type, 6	PERM READ WRITE, 40
type, o	PERM_READ_WRITE_EXEC, 40
ele_dtor_fn	PERM WRITE, 40
vec_st, 21	-
execute_command	rm, 52
shell.c, 249	touch, 54
EXITED NORMALLY	TYPE_DIRECTORY, 40
signal.h, 183	TYPE_REGULAR, 41
EXPECT COMMANDS	TYPE_SYMLINK, 41
parser.h, 242	TYPE_UNKNOWN, 41
EXPECT INPUT FILENAME	unmount, 55
parser.h, 242	fat_size
EXPECT_OUTPUT_FILENAME	fs_helpers.c, 69
	fs_helpers.h, 87
parser.h, 243	fd_entry_t, 7
F APPEND	filename, 7
fat_routines.h, 38	first_block, 7
F READ	in_use, 7
fat routines.h, 39	mode, 7
F WRITE	position, 8
fat routines.h, 39	ref_count, 8
fat	size, 8
fs_helpers.c, 69	fd_table
fs_helpers.h, 87	fs_helpers.c, 70
FAT EOF	fs_helpers.h, 87
fat_routines.h, 39	pcb_st, 13
FAT FREE	FILE_DESCRIPTOR_TABLE_SIZE
fat_routines.h, 39	kern_pcb.h, 130
fat_routines.c	filename
	fd_entry_t, 7
cat, 24	fill_buffer_until_full_or_newline
chmod, 28	shell.c, 251
cmpctdir, 29	find_file
cp, 29	

fs_helpers.c, 66	has_executable_permission, 84
fs_helpers.h, 82	increment_fd_ref_count, 85
FINISHED	init_fd_table, <mark>85</mark>
Job.h, 236	is_mounted, 88
finished_count	mark_entry_as_deleted, 86
job_st, 9	MAX_FDS, 88
first_block	num_fat_blocks, 88
fd_entry_t, 7	fs_kfuncs.c
firstBlock	current_fg_pid, 100
dir_entry_t, 5	current_running_pcb, 100
free_job_ptr	k_close, 89
shell.c, 251	k_ls, 90
free_pcb	k_lseek, 92
kern_pcb.c, 125	k_open, 92
kern_pcb.h, 131	k_read, 94
free scheduler queues	k_unlink, 96
scheduler.c, 165	k_write, 96
scheduler.h, 176	fs kfuncs.h
fs_fd	_ k_close, 102
fs_helpers.c, 70	k_ls, 102
fs_helpers.h, 87	k_lseek, 105
fs_helpers.c	k open, 106
add_file_entry, 57	k read, 108
allocate_block, 58	k_unlink, 110
block_size, 69	k write, 111
compact_directory, 59	SEEK CUR, 101
copy_host_to_pennfat, 62	SEEK END, 101
copy_pennfat_to_host, 63	SEEK_SET, 101
copy_source_to_dest, 64	fs_syscalls.c
decrement_fd_ref_count, 65	s_close, 115
fat, 69	s_ls, 116
	s_lseek, 116
fat_size, 69	s open, 116
fd_table, 70	— ·
find_file, 66	s_read, 117
fs_fd, 70	s_unlink, 117
get_free_fd, 67	s_write, 117
has_executable_permission, 67	fs_syscalls.h
increment_fd_ref_count, 67	s_close, 119
init_fd_table, 68	s_ls, 120
is_mounted, 70	s_lseek, 120
mark_entry_as_deleted, 68	s_open, 121
MAX_FDS, 70	s_read, 122
num_fat_blocks, 70	s_unlink, 123
fs_helpers.h	s_write, 123
add_file_entry, 72	STDERR_FILENO, 119
allocate_block, 74	STDIN_FILENO, 119
block_size, 87	STDOUT_FILENO, 119
compact_directory, 75	gonorato poyt priority
copy_host_to_pennfat, 77	generate_next_priority scheduler.c, 165
copy_pennfat_to_host, 79	
copy_source_to_dest, 80	scheduler.h, 176
decrement_fd_ref_count, 81	get_associated_ufunc
fat, 87	shell_built_ins.c, 270
fat_size, 87	get_free_fd
fd_table, 87	fs_helpers.c, 67
find_file, 82	fs_helpers.h, 83
fs_fd, 87	get_next_pcb
get_free_fd, 83	scheduler.c, 166
	scheduler.h, 177

get_pcb_in_queue	shell.c, 257
scheduler.c, 166	job_st, 8
scheduler.h, 178	cmd, 9
	finished_count, 9
handle_signal	id, 9
scheduler.c, 167	num_pids, 9
scheduler.h, 178	pgid, 9
hang	pids, 10
stress.c, 186	state, 10
stress.h, 187	job state t
has_executable_permission	Job.h, 236
fs_helpers.c, 67	JUMP_OUT
fs_helpers.h, 84	parser.c, 237
_ ,	pa. 551.6, 257
id	k_close
job_st, 9	fs_kfuncs.c, 89
in_use	fs_kfuncs.h, 102
fd_entry_t, 7	k_ls
increment_fd_ref_count	fs_kfuncs.c, 90
fs_helpers.c, 67	fs_kfuncs.h, 102
fs helpers.h, 85	k Iseek
init_fd_table	fs_kfuncs.c, 92
fs_helpers.c, 68	fs_kfuncs.h, 105
fs_helpers.h, 85	k_open
init_func	fs_kfuncs.c, 92
kern_sys_calls.c, 137	fs_kfuncs.h, 106
kern_sys_calls.h, 148	
initialize_scheduler_queues	k_proc_cleanup
·	kern_pcb.c, 126
scheduler.c, 167	kern_pcb.h, 132
scheduler.h, 179	k_proc_create
input_fd	kern_pcb.c, 127
pcb_st, 13	kern_pcb.h, 133
input_fd_script	k_read
shell.c, 256	fs_kfuncs.c, 94
is_append	fs_kfuncs.h, 108
shell.c, 256	k_unlink
shell.h, 259	fs_kfuncs.c, 96
is_background	fs_kfuncs.h, 110
parsed_command, 11	k_write
is_file_append	fs_kfuncs.c, 96
parsed_command, 11	fs_kfuncs.h, 111
is_mounted	kern_pcb.c
fs_helpers.c, 70	create pcb, 125
fs_helpers.h, 88	current_pcbs, 128
is_sleeping	current_running_pcb, 128
pcb_st, 13	free_pcb, 125
1 —	k_proc_cleanup, 126
jid_t	k_proc_create, 127
Job.h, 235	next_pid, 128
job	remove_child_in_parent, 127
Job.h, 236	_
Job.h	kern_pcb.h
FINISHED, 236	create_pcb, 130
jid_t, 235	FILE_DESCRIPTOR_TABLE_SIZE, 130
job, 236	free_pcb, 131
-	k_proc_cleanup, 132
job_state_t, 236	k_proc_create, 133
RUNNING, 236	pcb_t, 130
STOPPED, 236	remove_child_in_parent, 134
job_list	kern_sys_calls.c

current_fg_pid, 143		log_scheduling_event, 158
current_pcbs, 143		logger.h
current_running_pcb,		log_fd, 161
delete_from_explicit_c	•	log_generic_event, 159
delete_from_queue, 1		log_nice_event, 160
determine_index_in_c	queue, 136	log_scheduling_event, 161
init_func, 137	407	tick_counter, 161
move_pcb_correct_qu		ls
one_priority_queue, 1		fat_routines.c, 30
s_cleanup_init_proces	SS, 138	fat_routines.h, 48
s_echo, 138 s_exit, 139		main
s_exit, 139 s_kill, 139		pennfat.c, 227
s_nice, 139		pennos.c, 230
s_ps, 140		mark_entry_as_deleted
s_sleep, 140		fs_helpers.c, 68
s_spawn, 141		fs helpers.h, 86
s_spawn_init, 141		MAX BUFFER SIZE
s_waitpid, 142		shell.c, 248
sleep_blocked_queue	. 144	MAX_FDS
tick counter, 144	,	fs_helpers.c, 70
two_priority_queue, 1	44	fs_helpers.h, 88
zero_priority_queue,		MAX_LINE_BUFFER_SIZE
zombie_queue, 144		shell.c, 249
kern_sys_calls.h		meta
delete_from_explicit_d	queue, 146	spthread_st, 20
delete_from_queue, 1	47	meta_mutex
determine_index_in_c	queue, 147	spthread_meta_st, 17
init_func, 148		MILISEC_IN_NANO
move_pcb_correct_qu	ueue, 148	spthread.c, 195
s_cleanup_init_proces	ss, 149	mkfs
s_echo, 149		fat_routines.c, 30
s_exit, 150		fat_routines.h, 49
s_kill, 150		mode
s_nice, 151		fd_entry_t, 7
s_ps, 151		mount
s_sleep, 152		fat_routines.c, 32
s_spawn, 153		fat_routines.h, 50
s_spawn_init, 153		move_pcb_correct_queue
s_waitpid, 154		kern_sys_calls.c, 137 kern_sys_calls.h, 148
length		mtime
vec_st, 21		dir entry t, 5
log_fd		mv
logger.h, 161		fat_routines.c, 32
pennos.c, 230		fat routines.h, 51
scheduler.c, 171		lat_routines.n, or
log_generic_event		name
logger.c, 156		dir_entry_t, 6
logger.h, 159		next_job_id
log_nice_event		shell.c, 257
logger.c, 157		next_pid
logger.h, 160		kern_pcb.c, 128
log_scheduling_event		nohang
logger.c, 158		stress.c, 186
logger.h, 161		stress.h, 188
logger.c		num_commands
log_generic_event, 15	56	parsed_command, 11
log_nice_event, 157		num_fat_blocks

fs_helpers.c, 70	pennos-errno.h, 193
fs_helpers.h, 88	P EUNKNOWN
num_pids	pennos-errno.h, 193
job_st, 9	P EWRITE
,00 <u>-</u> 01, 0	pennos-errno.h, 193
one_priority_queue	•
	P_INITFAIL
kern_sys_calls.c, 144	pennos-errno.h, 193
scheduler.c, 171	P_NEEDF
orphan_child	pennos-errno.h, 194
shell_built_ins.c, 261	P_SIGCONT
output_fd	signal.h, 183
pcb_st, 14	P SIGSTOP
output_fd_script	signal.h, 184
shell.c, 257	P SIGTERM
, -	-
P EBADF	signal.h, 184
pennos-errno.h, 190	P_WIFEXITED
P EBUSY	signal.h, 184
— · · · · · · · · · · · · · · · · · · ·	P_WIFSIGNALED
pennos-errno.h, 190	signal.h, 184
P_ECLOSE	P WIFSTOPPED
pennos-errno.h, 190	signal.h, 184
P_ECOMMAND	par pid
pennos-errno.h, 190	pcb_st, 14
P EEXIST	
pennos-errno.h, 190	parse_command
P_EFS_NOT_MOUNTED	parser.c, 237
pennos-errno.h, 190	parser.h, 244
P_EFULL	parsed_command, 10
	commands, 11
pennos-errno.h, 191	is_background, 11
P_EFUNC	is_file_append, 11
pennos-errno.h, 191	num_commands, 11
P_EINTR	stdin_file, 11
pennos-errno.h, 191	stdout_file, 11
P EINVAL	
pennos-errno.h, 191	parser.c
P ELSEEK	JUMP_OUT, 237
pennos-errno.h, 191	parse_command, 237
P EMALLOC	print_parsed_command, 240
-	print_parser_errcode, 240
pennos-errno.h, 191	parser.h
P_EMAP	EXPECT COMMANDS, 242
pennos-errno.h, 192	EXPECT INPUT FILENAME, 242
P_ENOENT	EXPECT_OUTPUT_FILENAME, 243
pennos-errno.h, 192	parse_command, 244
P_ENULL	print_parsed_command, 246
pennos-errno.h, 192	
P EOPEN	print_parser_errcode, 247
pennos-errno.h, 192	UNEXPECTED_AMPERSAND, 243
P EPARSE	UNEXPECTED_FILE_INPUT, 243
-	UNEXPECTED_FILE_OUTPUT, 243
pennos-errno.h, 192	UNEXPECTED_PIPELINE, 243
P_EPERM	pcb_st, 12
pennos-errno.h, 192	child_pcbs, 13
P_EREAD	cmd_str, 13
pennos-errno.h, 193	fd_table, 13
P EREDIR	
pennos-errno.h, 193	input_fd, 13
P ERRNO	is_sleeping, 13
pennos-errno.c, 189	output_fd, 14
•	par_pid, 14
pennos-errno.h, 194	pid, 14
P_ESIGNAL	

priority, 14	PERM_WRITE
process state, 14	fat_routines.h, 40
process_status, 14	pgid
signals, 15	job_st, 9
thread_handle, 15	pid
time_to_wake, 15	pcb_st, 14
pcb_t	pids
kern_pcb.h, 130	job st, 10
_	position
pennfat.c	•
main, 227	fd_entry_t, 8
PROMPT, 227	print_parsed_command
pennos-errno.c	parser.c, 240
P_ERRNO, 189	parser.h, 246
pennos-errno.h	print_parser_errcode
P_EBADF, 190	parser.c, 240
P_EBUSY, 190	parser.h, 247
P_ECLOSE, 190	priority
P_ECOMMAND, 190	pcb_st, 14
P EEXIST, 190	process_state
P EFS NOT MOUNTED, 190	pcb st, 14
P EFULL, 191	process_status
P EFUNC, 191	pcb_st, 14
P EINTR, 191	PROMPT
P EINVAL, 191	-
-	pennfat.c, 227
P_ELSEEK, 191	shell.c, 249
P_EMALLOC, 191	pthread_fn
P_EMAP, 192	spthread.c, 197
P_ENOENT, 192	ptr_dtor_fn
P_ENULL, 192	Vec.h, 219
P_EOPEN, 192	ptr_t
P_EPARSE, 192	Vec.h, 219
P_EPERM, 192	put_pcb_into_correct_queue
P_EREAD, 193	scheduler.c, 168
P_EREDIR, 193	scheduler.h, 180
P ERRNO, 194	
P ESIGNAL, 193	recur
P_EUNKNOWN, 193	stress.c, 186
P EWRITE, 193	stress.h, 188
P INITFAIL, 193	ref_count
P NEEDF, 194	fd_entry_t, 8
pennos.c	remove_child_in_parent
log_fd, 230	kern_pcb.c, 127
•	kern_pcb.h, 134
main, 230	reserved
tick_counter, 230	
perm	dir_entry_t, 6
dir_entry_t, 6	rm
PERM_EXEC	fat_routines.c, 34
fat_routines.h, 39	fat_routines.h, 52
PERM_NONE	RUNNING
fat_routines.h, 39	Job.h, 236
PERM_READ	
fat_routines.h, 40	s_cleanup_init_process
PERM_READ_EXEC	kern_sys_calls.c, 138
fat routines.h, 40	kern_sys_calls.h, 149
PERM_READ_WRITE	s_close
fat routines.h, 40	fs_syscalls.c, 115
PERM_READ_WRITE_EXEC	fs_syscalls.h, 119
fat routines.h, 40	s_echo
iat_10utines.n, +0	kern_sys_calls.c, 138
	, , _

kern_sys_calls.h, 149 s_exit	delete_process_from_all_queues_except_current, 164
kern_sys_calls.c, 139	delete_process_from_particular_queue, 165
kern_sys_calls.h, 150	det_priorities_arr, 171
s_kill	free_scheduler_queues, 165
kern_sys_calls.c, 139	generate_next_priority, 165
kern_sys_calls.h, 150	get_next_pcb, 166
s_ls	get_pcb_in_queue, 166
fs_syscalls.c, 116	handle_signal, 167
fs_syscalls.h, 120	initialize_scheduler_queues, 167
s_lseek	log_fd, 171
fs_syscalls.c, 116	one_priority_queue, 171
fs_syscalls.h, 120	put_pcb_into_correct_queue, 168
s_nice	s_shutdown_pennos, 168
kern_sys_calls.c, 139	scheduler, 168
kern_sys_calls.h, 151	sleep_blocked_queue, 171
s_open	tick_counter, 171
fs_syscalls.c, 116	two_priority_queue, 171
fs_syscalls.h, 121	zero_priority_queue, 172
s_ps	zombie_queue, 172
kern_sys_calls.c, 140	scheduler.h
kern_sys_calls.h, 151	alarm_handler, 173
s_read	child_in_zombie_queue, 174
fs_syscalls.c, 117	child_with_changed_process_status, 174
fs_syscalls.h, 122	delete_process_from_all_queues, 175
s_shutdown_pennos	delete_process_from_all_queues_except_current,
scheduler.c, 168	175
scheduler.h, 180	delete_process_from_particular_queue, 176
s_sleep	free_scheduler_queues, 176
kern_sys_calls.c, 140	generate_next_priority, 176
kern_sys_calls.h, 152	get_next_pcb, 177
s_spawn	get_pcb_in_queue, 178
kern_sys_calls.c, 141	handle_signal, 178
kern_sys_calls.h, 153	initialize_scheduler_queues, 179
s_spawn_init	put_pcb_into_correct_queue, 180
kern_sys_calls.c, 141	s_shutdown_pennos, 180
kern_sys_calls.h, 153	scheduler, 180
s_unlink	script_fd
fs_syscalls.c, 117	shell.c, 257
fs_syscalls.h, 123	SEEK_CUR
s_waitpid	fs_kfuncs.h, 101
kern_sys_calls.c, 142	SEEK_END
kern_sys_calls.h, 154	fs_kfuncs.h, 101
s_write	SEEK_SET
fs_syscalls.c, 117	fs_kfuncs.h, 101
fs_syscalls.h, 123	setup_cond
scheduler	spthread_fwd_args_st, 16
scheduler.c, 168	setup_done
scheduler.h, 180	spthread_fwd_args_st, 16
scheduler.c	setup_mutex
alarm_handler, 163	spthread_fwd_args_st, 17
child_in_zombie_queue, 163	setup_terminal_signal_handlers
child_with_changed_process_status, 164	shell.c, 252
curr_priority_arr_index, 170	shell
current_pcbs, 170	shell.c, 252
current_running_pcb, 170	shell.h, 258
delete_process_from_all_queues, 164	shell.c
	current_fg_pid, 256

execute_command, 249	u_man, <mark>277</mark>
fill_buffer_until_full_or_newline, 251	u_mv, 277
free_job_ptr, 251	u_nice, 278
input_fd_script, 256	u_nice_pid, 278
is_append, 256	u_orphanify, 279
job_list, 257	u_ps, 279
MAX_BUFFER_SIZE, 248	u_rm, 279
MAX_LINE_BUFFER_SIZE, 249	u_sleep, 280
next_job_id, 257	u_touch, 280
	
output_fd_script, 257	u_zombify, 281
PROMPT, 249	shell_sigint_handler
script_fd, 257	shell.c, 254
setup_terminal_signal_handlers, 252	shell_sigstp_handler
shell, 252	shell.c, 254
shell_sigint_handler, 254	shutup_mutex
shell_sigstp_handler, 254	spthread_signal_args_st, 18
u_execute_command, 254	signal
u_read_and_execute_script, 255	spthread_signal_args_st, 19
shell.h	signal.h
is_append, 259	CONT_BY_SIG, 183
shell, 258	EXITED NORMALLY, 183
shell built ins.c	P_SIGCONT, 183
	P_SIGSTOP, 184
get_associated_ufunc, 270	
orphan_child, 261	P_SIGTERM, 184
u_bg, 261	P_WIFEXITED, 184
u_busy, 262	P_WIFSIGNALED, 184
u_cat, 262	P_WIFSTOPPED, 184
u_chmod, 262	STOPPED_BY_SIG, 185
u_cp, 263	TERM_BY_SIG, 185
<u></u>	′
u_echo, 263	signals
u_echo, 263	
u_echo, 263 u_fg, 264	signals
u_echo, 263 u_fg, 264 u_jobs, 264	signals pcb_st, 15 SIGPTHD
u_echo, 263 u_fg, 264 u_jobs, 264 u_kill, 264	signals pcb_st, 15 SIGPTHD spthread.h, 203
u_echo, 263 u_fg, 264 u_jobs, 264 u_kill, 264 u_logout, 265	signals pcb_st, 15 SIGPTHD spthread.h, 203 size
u_echo, 263 u_fg, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6
u_echo, 263 u_fg, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8
u_echo, 263 u_fg, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue
u_echo, 263 u_fg, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144
u_echo, 263 u_fg, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.c
u_echo, 263 u_fg, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.c
u_echo, 263 u_fg, 264 u_jobs, 264 u_jil, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.c _GNU_SOURCE, 195
u_echo, 263 u_fg, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.c _GNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_rm, 268 u_sleep, 269 u_touch, 269	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.c _GNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_mice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_rm, 268 u_sleep, 269 u_touch, 269 u_zombify, 270	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.c _GNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_continue, 197
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_rm, 268 u_sleep, 269 u_touch, 269 u_zombify, 270 zombie_child, 270	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.c _GNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_continue, 197 spthread_create, 198
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_rm, 268 u_sleep, 269 u_touch, 269 u_zombify, 270 zombie_child, 270 shell_built_ins.h	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.c _GNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_continue, 197 spthread_create, 198 spthread_disable_interrupts_self, 199
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_mice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_rm, 268 u_sleep, 269 u_touch, 269 u_zombify, 270 zombie_child, 270 shell_built_ins.h u_bg, 272	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.c _GNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_continue, 197 spthread_create, 198 spthread_disable_interrupts_self, 199 spthread_enable_interrupts_self, 199
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_mice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_rm, 268 u_sleep, 269 u_touch, 269 u_zombify, 270 zombie_child, 270 shell_built_ins.h u_bg, 272 u_busy, 273	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.c _GNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_continue, 197 spthread_create, 198 spthread_disable_interrupts_self, 199 spthread_equal, 200
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_rm, 268 u_sleep, 269 u_touch, 269 u_zombify, 270 zombie_child, 270 shell_built_ins.h u_bg, 272 u_busy, 273 u_cat, 273	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.c _GNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_continue, 197 spthread_create, 198 spthread_disable_interrupts_self, 199 spthread_enable_interrupts_self, 199 spthread_equal, 200 spthread_exit, 200
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_rm, 268 u_rm, 269 u_sleep, 269 u_touch, 269 u_zombify, 270 zombie_child, 270 shell_built_ins.h u_bg, 272 u_busy, 273 u_cat, 273 u_chmod, 273	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.cGNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_continue, 197 spthread_create, 198 spthread_create, 198 spthread_disable_interrupts_self, 199 spthread_enable_interrupts_self, 199 spthread_equal, 200 spthread_exit, 200 spthread_fwd_args, 197
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_rm, 268 u_sleep, 269 u_touch, 269 u_zombify, 270 zombie_child, 270 shell_built_ins.h u_bg, 272 u_busy, 273 u_cat, 273 u_chmod, 273 u_cp, 274	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.cGNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_continue, 197 spthread_create, 198 spthread_create, 198 spthread_disable_interrupts_self, 199 spthread_enable_interrupts_self, 199 spthread_equal, 200 spthread_exit, 200 spthread_fwd_args, 197 spthread_join, 200
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_rm, 268 u_sleep, 269 u_touch, 269 u_zombify, 270 zombie_child, 270 shell_built_ins.h u_bg, 272 u_busy, 273 u_cat, 273 u_chmod, 273 u_cp, 274 u_echo, 274	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.cGNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_continue, 197 spthread_create, 198 spthread_create, 198 spthread_disable_interrupts_self, 199 spthread_enable_interrupts_self, 199 spthread_equal, 200 spthread_exit, 200 spthread_fwd_args, 197 spthread_join, 200 spthread_meta_t, 197
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_rm, 268 u_sleep, 269 u_touch, 269 u_zombify, 270 zombie_child, 270 shell_built_ins.h u_bg, 272 u_busy, 273 u_cat, 273 u_chmod, 273 u_cp, 274	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.cGNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_continue, 197 spthread_create, 198 spthread_create, 198 spthread_disable_interrupts_self, 199 spthread_enable_interrupts_self, 199 spthread_equal, 200 spthread_exit, 200 spthread_fwd_args, 197 spthread_join, 200
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_rm, 268 u_sleep, 269 u_touch, 269 u_zombify, 270 zombie_child, 270 shell_built_ins.h u_bg, 272 u_busy, 273 u_cat, 273 u_chmod, 273 u_cp, 274 u_echo, 274	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.cGNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_continue, 197 spthread_create, 198 spthread_create, 198 spthread_disable_interrupts_self, 199 spthread_enable_interrupts_self, 199 spthread_equal, 200 spthread_exit, 200 spthread_fwd_args, 197 spthread_join, 200 spthread_meta_t, 197
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_rm, 268 u_sleep, 269 u_touch, 269 u_zombify, 270 zombie_child, 270 shell_built_ins.h u_bg, 272 u_busy, 273 u_cat, 273 u_chmod, 273 u_cp, 274 u_echo, 274 u_fg, 275	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.cGNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_continue, 197 spthread_create, 198 spthread_create, 198 spthread_disable_interrupts_self, 199 spthread_enable_interrupts_self, 199 spthread_equal, 200 spthread_exit, 200 spthread_fwd_args, 197 spthread_join, 200 spthread_meta_t, 197 SPTHREAD_RUNNING_STATE, 196
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_sleep, 269 u_touch, 269 u_zombify, 270 zombie_child, 270 shell_built_ins.h u_bg, 272 u_busy, 273 u_cat, 273 u_chmod, 273 u_cp, 274 u_echo, 274 u_fg, 275 u_jobs, 275 u_kill, 275	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.cGNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_create, 198 spthread_create, 198 spthread_disable_interrupts_self, 199 spthread_enable_interrupts_self, 199 spthread_equal, 200 spthread_exit, 200 spthread_fwd_args, 197 spthread_join, 200 spthread_meta_t, 197 SPTHREAD_RUNNING_STATE, 196 spthread_self, 200
u_echo, 263 u_fg, 264 u_jobs, 264 u_jobs, 264 u_kill, 264 u_logout, 265 u_ls, 265 u_man, 266 u_mv, 266 u_nice, 267 u_nice_pid, 267 u_orphanify, 268 u_ps, 268 u_rm, 268 u_sleep, 269 u_touch, 269 u_zombify, 270 zombie_child, 270 shell_built_ins.h u_bg, 272 u_busy, 273 u_cat, 273 u_chmod, 273 u_cp, 274 u_fg, 275 u_jobs, 275	signals pcb_st, 15 SIGPTHD spthread.h, 203 size dir_entry_t, 6 fd_entry_t, 8 sleep_blocked_queue kern_sys_calls.c, 144 scheduler.c, 171 spthread.cGNU_SOURCE, 195 MILISEC_IN_NANO, 195 pthread_fn, 197 spthread_cancel, 197 spthread_create, 198 spthread_create, 198 spthread_disable_interrupts_self, 199 spthread_enable_interrupts_self, 199 spthread_equal, 200 spthread_exit, 200 spthread_fwd_args, 197 spthread_meta_t, 197 SPTHREAD_RUNNING_STATE, 196 spthread_self, 200 SPTHREAD_SIG_CONTINUE, 196

spthread_suspend, 201	spthread.h, 203
spthread_suspend_self, 201	SPTHREAD_RUNNING_STATE
SPTHREAD_SUSPENDED_STATE, 196	spthread.c, 196
SPTHREAD_TERMINATED_STATE, 196	spthread_self
spthread.h	spthread.c, 200
SIGPTHD, 203	spthread.h, 207
spthread_cancel, 204	SPTHREAD_SIG_CONTINUE
spthread_continue, 204	spthread.c, 196
spthread_create, 205	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, 207	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, 201
spthread.h, 204	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, 205	spthread_t
spthread_disable_interrupts_self	spthread.h, 203
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, 37
spthread.h, 206	SRC/fs/fs_helpers.c, 56
spthread equal	SRC/fs/fs_helpers.h, 71
spthread.c, 200	SRC/fs/fs_kfuncs.c, 88
spthread.h, 206	SRC/fs/fs_kfuncs.h, 100
spthread_exit	SRC/fs/fs syscalls.c, 114
spthread.c, 200	SRC/fs/fs_syscalls.h, 118
spthread.h, 206	SRC/kernel/kern_pcb.c, 124
spthread fwd args	SRC/kernel/kern pcb.h, 129
spthread.c, 197	SRC/kernel/kern sys calls.c, 134
spthread_fwd_args_st, 15	SRC/kernel/kern sys calls.h, 145
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, 207	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, 209
spthread_meta_t	SRC/lib/Vec.h, 217
spthread.c, 197	SRC/pennfat.c, 227

SRC/pennos.c, 229	type
SRC/shell/builtins.c, 231	dir_entry_t, 6
SRC/shell/builtins.h, 233	TYPE DIRECTORY
SRC/shell/Job.h, 234	fat routines.h, 40
SRC/shell/parser.c, 236	TYPE REGULAR
SRC/shell/parser.h, 241	fat_routines.h, 41
SRC/shell/shell.c, 247	TYPE SYMLINK
SRC/shell/shell.h, 257	fat routines.h, 41
SRC/shell/shell_built_ins.c, 260	TYPE UNKNOWN
SRC/shell/shell_built_ins.h, 271	fat_routines.h, 41
state	iat_routinos.n, Tr
job_st, 10	u_bg
spthread_meta_st, 17	shell_built_ins.c, 261
STDERR FILENO	shell_built_ins.h, 272
fs_syscalls.h, 119	u_busy
stdin_file	shell_built_ins.c, 262
	shell_built_ins.h, 273
parsed_command, 11	u cat
STDIN_FILENO	shell_built_ins.c, 262
fs_syscalls.h, 119	shell_built_ins.h, 273
stdout_file	u chmod
parsed_command, 11	shell_built_ins.c, 262
STDOUT_FILENO	shell built ins.h, 273
fs_syscalls.h, 119	
STOPPED	u_cp
Job.h, 236	shell_built_ins.c, 263
STOPPED_BY_SIG	shell_built_ins.h, 274
signal.h, 185	u_echo
stress.c	shell_built_ins.c, 263
crash, 186	shell_built_ins.h, 274
hang, 186	u_execute_command
nohang, 186	shell.c, 254
recur, 186	u_fg
stress.h	shell_built_ins.c, 264
crash, 187	shell_built_ins.h, 275
hang, 187	u_jobs
nohang, 188	shell_built_ins.c, 264
recur, 188	shell_built_ins.h, 275
suspend_set	u_kill
spthread_meta_st, 18	shell_built_ins.c, 264
spinicad_neta_st, 10	shell built ins.h, 275
TERM_BY_SIG	u logout
 signal.h, 185	shell_built_ins.c, 265
thread	shell built ins.h, 276
spthread_st, 20	u_ls
thread handle	shell_built_ins.c, 265
pcb_st, 15	shell built ins.h, 276
tick_counter	u_man
kern_sys_calls.c, 144	shell_built_ins.c, 266
logger.h, 161	shell_built_ins.h, 277
pennos.c, 230	
•	u_mv
scheduler.c, 171	shell_built_ins.c, 266
time_to_wake	shell_built_ins.h, 277
pcb_st, 15	u_nice
touch	shell_built_ins.c, 267
fat_routines.c, 35	shell_built_ins.h, 278
fat_routines.h, 54	u_nice_pid
two_priority_queue	shell_built_ins.c, 267
kern_sys_calls.c, 144	shell_built_ins.h, 278
scheduler.c, 171	u_orphanify

shell_built_ins.c, 268	vec_is_empty, 218
shell_built_ins.h, 279	vec_len, 218
u_perror	vec_new, 223
builtins.c, 231	vec_pop_back, 224
builtins.h, 233	vec_push_back, 224
u_ps	vec_resize, 225
shell_built_ins.c, 268	vec_set, 226
shell_built_ins.h, 279	vec_capacity
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, 279	vec_destroy
u_sleep	Vec.c, 210
shell_built_ins.c, 269	Vec.h, 220
shell_built_ins.h, 280	vec_erase
u_touch	Vec.c, 210
shell_built_ins.c, 269	Vec.h, 220
shell_built_ins.h, 280	vec_erase_no_deletor
u_zombify	Vec.c, 211
shell_built_ins.c, 270	Vec.h, 221
shell_built_ins.h, 281	vec_get
UNEXPECTED_AMPERSAND	Vec.c, 212
parser.h, 243	Vec.h, 222
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 fat_routines.h, 55	vec_new
lat_routilles.fr, 55	Vec.c, 213 Vec.h, 223
Vec	vec_pop_back
Vec.h, 219	Vec_pop_back Vec.c, 214
Vec.c	•
vec_clear, 209	Vec.h, 224
vec_destroy, 210	vec_push_back Vec.c, 214
vec_erase, 210	Vec.h, 224
vec_erase_no_deletor, 211	vec_resize
vec_get, 212	Vec.c, 215
vec insert, 212	Vec.h, 225
vec_new, 213	vec_set
vec_pop_back, 214	Vec.c, 216
vec_push_back, 214	Vec.h, 226
vec resize, 215	vec st, 20
vec set, 216	capacity, 20
Vec.h	data, 21
ptr_dtor_fn, 219	ele_dtor_fn, 21
ptr_t, 219	length, 21
Vec, 219	iongin, Zi
vec_capacity, 218	zero_priority_queue
vec_clear, 219	kern_sys_calls.c, 144
vec_destroy, 220	scheduler.c, 172
vec_erase, 220	zombie_child
vec_erase_no_deletor, 221	shell_built_ins.c, 270
vec_get, 222	zombie_queue
vec_insert, 222	kern_sys_calls.c, 144
- .	

scheduler.c, 172