PennOS

Generated by Doxygen 1.9.1

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

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

3.9.1 Detailed Description	19
3.9.2 Member Data Documentation	20
3.9.2.1 meta	20
3.9.2.2 thread	20
3.10 vec_st Struct Reference	20
3.10.1 Detailed Description	20
3.10.2 Member Data Documentation	20
3.10.2.1 capacity	21
3.10.2.2 data	21
3.10.2.3 ele_dtor_fn	21
3.10.2.4 length	21
4 File Documentation	23
4.1 SRC/fs/fat_routines.c File Reference	23
4.1.1 Function Documentation	
4.1.1.1 cat()	24
4.1.1.2 chmod()	28
4.1.1.3 cmpctdir()	29
4.1.1.4 cp()	29
4.1.1.5 ls()	30
4.1.1.6 mkfs()	31
4.1.1.7 mount()	32
4.1.1.8 mv()	33
4.1.1.9 rm()	34
4.1.1.10 touch()	35
4.1.1.11 unmount()	36
4.1.2 Variable Documentation	37
4.1.2.1 current_running_pcb	37
4.2 SRC/fs/fat_routines.h File Reference	37
4.2.1 Macro Definition Documentation	39
4.2.1.1 F_APPEND	39
4.2.1.2 F_READ	39
4.2.1.3 F_WRITE	39
4.2.1.4 FAT_EOF	39
4.2.1.5 FAT_FREE	39
4.2.1.6 PERM_EXEC	40
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	41

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	. 70
4.3.2.1 block_size	. 70
4.3.2.2 fat	. 70
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	. 71
4.3.2.7 MAX_FDS	. 71
4.3.2.8 num_fat_blocks	. 71
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()	. 78
4.4.1.5 copy_pennfat_to_host()	. 79
4.4.1.6 copy_source_to_dest()	. 81
4.4.1.7 decrement_fd_ref_count()	. 82
4.4.1.8 find_file()	. 83
4.4.1.9 get_free_fd()	. 84
4.4.1.10 has_executable_permission()	. 84
4.4.1.11 increment_fd_ref_count()	. 85
4.4.1.12 init_fd_table()	. 86
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	. 88
4.4.2.4 fd_table	. 88
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	. 89
4.5.1 Function Documentation	. 90
4.5.1.1 k_close()	. 90
4.5.1.2 k_ls()	. 90
4.5.1.3 k_lseek()	. 92
4.5.1.4 k_open()	. 93
4.5.1.5 k_read()	. 95
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	. 101
4.6.1 Macro Definition Documentation	. 102
4.6.1.1 SEEK_CUR	. 102
4.6.1.2 SEEK_END	. 102
4.6.1.3 SEEK_SET	. 102
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() 168
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()	180
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()	230
4.28.2 Variable Documentation	230
4.28.2.1 log_fd	230
4.28.2.2 tick_counter	231
4.29 SRC/shell/builtins.c File Reference	231
4.29.1 Function Documentation	231
4.29.1.1 u_perror()	231
4.30 SRC/shell/builtins.h File Reference	233
4.30.1 Function Documentation	233
4.30.1.1 u_perror()	233
4.31 SRC/shell/Job.h File Reference	234
4.31.1 Typedef Documentation	236
4.31.1.1 jid_t	236
4.31.1.2 job	236
4.31.2 Enumeration Type Documentation	236
4.31.2.1 job_state_t	236
4.32 SRC/shell/parser.c File Reference	236
4.32.1 Macro Definition Documentation	237
4.32.1.1 JUMP_OUT	237
4.32.2 Function Documentation	237
4.32.2.1 parse_command()	238
4.32.2.2 print_parsed_command()	240
4.32.2.3 print_parser_errcode()	241
4.33 SRC/shell/parser.h File Reference	241
4.33.1 Macro Definition Documentation	242
4.33.1.1 EXPECT_COMMANDS	242
4.33.1.2 EXPECT_INPUT_FILENAME	243
4.33.1.3 EXPECT_OUTPUT_FILENAME	243
4.33.1.4 UNEXPECTED_AMPERSAND	243
4.33.1.5 UNEXPECTED_FILE_INPUT	243
4.33.1.6 UNEXPECTED_FILE_OUTPUT	243
4.33.1.7 UNEXPECTED_PIPELINE	243
4.33.2 Function Documentation	244
4.33.2.1 parse_command()	244
4.33.2.2 print_parsed_command()	246
4.33.2.3 print_parser_errcode()	247
4.34 SRC/shell/shell.c File Reference	247
4.34.1 Macro Definition Documentation	248
4.34.1.1 MAX_BUFFER_SIZE	249
4.34.1.2 MAX_LINE_BUFFER_SIZE	249
4.34.1.3 PROMPT	249
4.34.2 Function Documentation	DΛQ

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()	. 255
4.34.2.7 shell_sigstp_handler()	. 255
4.34.2.8 u_execute_command()	. 255
4.34.2.9 u_read_and_execute_script()	. 256
4.34.3 Variable Documentation	. 257
4.34.3.1 current_fg_pid	. 257
4.34.3.2 input_fd_script	. 257
4.34.3.3 is_append	. 258
4.34.3.4 job_list	. 258
4.34.3.5 next_job_id	. 258
4.34.3.6 output_fd_script	. 258
4.34.3.7 script_fd	. 258
4.35 SRC/shell/shell.h File Reference	. 258
4.35.1 Function Documentation	. 259
4.35.1.1 shell()	. 259
4.35.2 Variable Documentation	. 261
4.35.2.1 is_append	. 262
4.36 SRC/shell/shell_built_ins.c File Reference	. 262
4.36.1 Function Documentation	. 263
4.36.1.1 findJobByIdOrCurrent()	. 264
4.36.1.2 orphan_child()	. 264
4.36.1.3 u_bg()	. 265
4.36.1.4 u_busy()	. 265
4.36.1.5 u_cat()	. 266
4.36.1.6 u_chmod()	. 266
4.36.1.7 u_cp()	. 266
4.36.1.8 u_echo()	. 267
4.36.1.9 u_fg()	. 267
4.36.1.10 u_jobs()	. 268
4.36.1.11 u_kill()	. 269
4.36.1.12 u_logout()	. 270
4.36.1.13 u_ls()	. 270
4.36.1.14 u_man()	. 270
4.36.1.15 u_mv()	. 271
4.36.1.16 u_nice()	. 271
4.36.1.17 u_nice_pid()	. 272
4.36.1.18 u_orphanify()	. 272

289

4.36.1.19 u_ps()	 . 272
4.36.1.20 u_rm()	 . 273
4.36.1.21 u_sleep()	 . 273
4.36.1.22 u_touch()	 . 273
4.36.1.23 u_zombify()	 . 274
4.36.1.24 zombie_child()	 . 274
4.36.2 Variable Documentation	 . 274
4.36.2.1 current_fg_pid	 . 274
4.36.2.2 get_associated_ufunc	 . 274
4.36.2.3 job_list	 . 275
4.37 SRC/shell/shell_built_ins.h File Reference	 . 275
4.37.1 Function Documentation	 . 277
4.37.1.1 findJobByIdOrCurrent()	 . 277
4.37.1.2 u_bg()	 . 278
4.37.1.3 u_busy()	 . 279
4.37.1.4 u_cat()	 . 279
4.37.1.5 u_chmod()	 . 280
4.37.1.6 u_cp()	 . 280
4.37.1.7 u_echo()	 . 281
4.37.1.8 u_fg()	 . 281
4.37.1.9 u_jobs()	 . 282
4.37.1.10 u_kill()	 . 283
4.37.1.11 u_logout()	 . 284
4.37.1.12 u_ls()	 . 284
4.37.1.13 u_man()	 . 284
4.37.1.14 u_mv()	 . 285
4.37.1.15 u_nice()	 . 286
4.37.1.16 u_nice_pid()	 . 286
4.37.1.17 u_orphanify()	 . 287
4.37.1.18 u_ps()	 . 287
4.37.1.19 u_rm()	 . 287
4.37.1.20 u_sleep()	 . 288
4.37.1.21 u_touch()	 . 288
4.37.1.22 u_zombify()	 . 288

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 47 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 50 of file fat_routines.h.

3.1.2.2 mtime

```
time_t dir_entry_t::mtime
```

Definition at line 53 of file fat_routines.h.

3.1.2.3 name

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

Definition at line 48 of file fat_routines.h.

3.1.2.4 perm

```
uint8_t dir_entry_t::perm
```

Definition at line 52 of file fat_routines.h.

3.1.2.5 reserved

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

Definition at line 54 of file fat_routines.h.

3.1.2.6 size

```
uint32_t dir_entry_t::size
```

Definition at line 49 of file fat_routines.h.

3.1.2.7 type

```
uint8_t dir_entry_t::type
```

Definition at line 51 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 60 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 63 of file fat_routines.h.

3.2.2.2 first_block

```
uint16_t fd_entry_t::first_block
```

Definition at line 65 of file fat_routines.h.

3.2.2.3 in_use

```
int fd_entry_t::in_use
```

Definition at line 61 of file fat_routines.h.

3.2.2.4 mode

```
uint8_t fd_entry_t::mode
```

Definition at line 67 of file fat_routines.h.

3.2.2.5 position

```
uint32_t fd_entry_t::position
```

Definition at line 66 of file fat_routines.h.

3.2.2.6 ref_count

```
int fd_entry_t::ref_count
```

Definition at line 62 of file fat_routines.h.

3.2.2.7 size

```
uint32_t fd_entry_t::size
```

Definition at line 64 of file fat_routines.h.

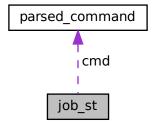
The documentation for this struct was generated from the following file:

• 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:

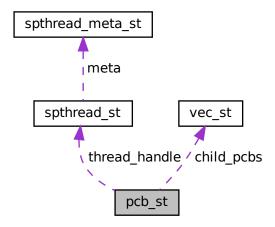
· SRC/shell/parser.h

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 29 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 35 of file kern_pcb.h.

3.5.2.2 cmd_str

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

Definition at line 41 of file kern_pcb.h.

3.5.2.3 fd_table

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

Definition at line 60 of file kern_pcb.h.

3.5.2.4 input_fd

```
int pcb_st::input_fd
```

Definition at line 47 of file kern_pcb.h.

3.5.2.5 is_sleeping

```
bool pcb_st::is_sleeping
```

Definition at line 57 of file kern_pcb.h.

3.5.2.6 output_fd

```
int pcb_st::output_fd
```

Definition at line 48 of file kern_pcb.h.

3.5.2.7 par_pid

```
pid_t pcb_st::par_pid
```

Definition at line 33 of file kern_pcb.h.

3.5.2.8 pid

```
pid_t pcb_st::pid
```

Definition at line 32 of file kern_pcb.h.

3.5.2.9 priority

```
int pcb_st::priority
```

Definition at line 37 of file kern_pcb.h.

3.5.2.10 process_state

char pcb_st::process_state

Definition at line 38 of file kern_pcb.h.

3.5.2.11 process_status

```
int pcb_st::process_status
```

Definition at line 50 of file kern_pcb.h.

3.5.2.12 signals

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

Definition at line 43 of file kern_pcb.h.

3.5.2.13 thread_handle

```
spthread_t pcb_st::thread_handle
```

Definition at line 30 of file kern_pcb.h.

3.5.2.14 time_to_wake

```
int pcb_st::time_to_wake
```

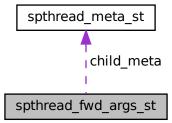
Definition at line 58 of file kern_pcb.h.

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.

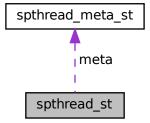
The documentation for this struct was generated from the following file:

• 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

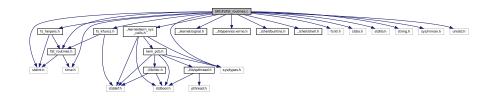
22 Class Documentation

Chapter 4

File Documentation

4.1 SRC/fs/fat_routines.c File Reference

```
#include "fat_routines.h"
#include "../kernel/kern_sys_calls.h"
#include "../kernel/signal.h"
#include "../lib/pennos-errno.h"
#include "../shell/builtins.h"
#include "../shell/shell.h"
#include "fs_helpers.h"
#include "fs_kfuncs.h"
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
Include dependency graph for fat_routines.c:
```



Functions

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

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

• int unmount ()

Unmounts the current filesystem and reset variables.

void * cat (void *arg)

Concatenates and displays files.

void * 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
217
      \ensuremath{//} verify that the file system is mounted
      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 we're
235
           // appending
           if ((strcmp(file_1, file_2) == 0) && is_append) {
236
            P_ERRNO = P_EREDIR;
u_perror("cat");
237
238
239
             return NULL;
240
241
           // edge case when input and output files names are the same but we're not // appending truncates the file \,
242
243
           if ((strcmp(file_1, file_2) == 0)) {
245
            return NULL;
246
247
           \ensuremath{//} 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);
252
             u_perror("cat");
253
             return NULL;
2.54
           if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
255
256
             k_close(in_fd);
             u_perror("cat");
258
             return NULL;
259
260
261
           char* buffer = (char*)malloc(block size);
           if (buffer == NULL) {
   P_ERRNO = P_EMALLOC;
262
263
264
             k_close(in_fd);
265
             u_perror("cat");
266
             return NULL;
2.67
268
269
           int bytes_read;
270
           ssize_t bytes_remaining = in_fd_size;
271
272
           while (bytes_remaining > 0) {
273
           ssize_t bytes_to_read =
                 bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
274
            bytes_read = k_read(in_fd, buffer, bytes_to_read);
275
276
277
             if (bytes_read <= 0) {</pre>
            break;
278
279
280
             if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
281
                free (buffer);
283
                k_close(in_fd);
```

u_perror("cat");

284

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

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

```
459
         k_close(in_fd);
460
        free (buffer);
461
462
      // close output file if not stdout
if (out_fd != STDOUT_FILENO) {
463
464
465
       k_close(out_fd);
466
467
468
       return NULL;
469 }
```

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 770 of file fat_routines.c.

```
771
772
        char** args = (char**)arg;
        if (!args || !args[0] || !args[1] || !args[2]) {
    P_ERRNO = P_EINVAL;
773
774
          return NULL;
775
776
777
        // Parse permission string
778
779
        const char* perm_str = args[1];
if (perm_str[0] != '+' && perm_str[0] != '-') {
  P_ERRNO = P_EINVAL;
return NULL;
780
          return NULL;
781
782
783
       \ensuremath{//} Find the file and get its current directory entry
784
       dir_entry_t dir_entry;
int entry_offset = find_file(args[2], &dir_entry);
if (entry_offset < 0) {
   P_ERRNO = P_ENOENT;</pre>
785
786
787
788
789
          return NULL;
790
791
        // Calculate new permissions
792
       uint8_t new_perm = dir_entry.perm;
int i = 1; // Start after + or -
while (perm_str[i] != '\0') {
793
794
795
          switch (perm_str[i]) {
   case 'r':
    if (perm_str[0] == '+') {
796
797
798
799
                  new_perm |= PERM_READ;
              } else {
800
801
                  new_perm &= ~PERM_READ;
802
             break;
case 'w':
  if (perm_str[0] == '+') {
803
804
805
                  new_perm |= PERM_WRITE;
807
               } else {
808
                  new_perm &= ~PERM_WRITE;
             break;
case 'x':
809
810
811
812
              if (perm_str[0] == '+') {
                   new_perm |= PERM_EXEC;
```

```
} else {
815
                 new_perm &= ~PERM_EXEC;
816
817
               break;
818
             default:
             P_ERRNO = P_EINVAL;
return NULL;
819
820
821
822
823
       }
824
       // Update the directory entry
dir_entry.perm = new_perm;
dir_entry.mtime = time(NULL);
825
826
827
       // Seek to the entry's position
if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
   return NULL.
828
829
830
831
832
          return NULL;
833
834
        // Write the updated entry back
835
       if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
   P_ERRNO = P_EWRITE;
836
837
838
          return NULL;
839 }
840
841 return NULL;
842 }
```

4.1.1.3 cmpctdir()

Implements compaction of root directory.

Compacts the root directory by removing all deleted entries.

Definition at line 851 of file fat_routines.c.

4.1.1.4 cp()

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

Copies the source file to the destination.

Copies files.

Definition at line 640 of file fat_routines.c.

```
641
      char** args = (char**)arg;
642
643
      // check that we have enough arguments
      if (args[1] == NULL | | args[2] == NULL) {
   P_ERRNO = P_EINVAL;
644
645
        u_perror("cp");
646
647
        return NULL;
648
649
      // cp -h SOURCE DEST
if (strcmp(args[1], "-h") == 0) {
  if (args[2] == NULL || args[3] == NULL) {
    P_ERRNO = P_EINVAL;
    u_perror("cp");
650
651
652
653
654
655
          return NULL;
656
657
        if (copy_host_to_pennfat(args[2], args[3]) != 0) {
  u_perror("cp");
658
659
660
          return NULL;
661
662
        return NULL;
663
      }
664
665
      // cp SOURCE -h DEST
      if (args[2] != NULL && strcmp(args[2], "-h") == 0) {
666
667
       if (args[3] == NULL) {
         P_ERRNO = P_EINVAL;
u_perror("cp");
668
669
          return NULL;
670
671
672
673
        if (copy_pennfat_to_host(args[1], args[3]) != 0) {
        u_perror("cp");
return NULL;
674
675
676
677
        return NULL;
678
679
680
      // cp SOURCE DEST
      681
682
683
684
685
686
        }
687
        return NULL;
688
      }
689
690
      P_ERRNO = P_EUNKNOWN;
      u_perror("cp");
692
      return NULL;
693 }
```

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 476 of file fat_routines.c.

```
476
477

// Note: we already check if fs is mounted in k_ls
478
479
char** args = (char**) arg;
480
if (args[1] != NULL) {
481
if (k_ls(args[1]) == -1) {
482
u_perror("ls");
783
return NULL;
484
}
```

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.

```
34
      // validate arguments
if (num_blocks < 1 || num_blocks > 32) {
  P_ERRNO = P_EINVAL;
35
37
38
         return -1;
39
      if (blk_size < 0 || blk_size > 4) {
40
41
         P_ERRNO = P_EINVAL;
         return -1;
42
44
      // determine the file system size
int block_sizes[] = {256, 512, 1024, 2048, 4096};
int actual_block_size = block_sizes[blk_size];
4.5
46
47
      int fat_size = num_blocks * actual_block_size;
48
49
      int fat_entries = fat_size / 2;
50
      int num_data_blocks =
51
             (num\_blocks == 32)
      ? fat_entries - 2

: fat_entries - 1; // note: first entry is reserved for metadata!

size_t filesystem_size = fat_size + (actual_block_size * num_data_blocks);
52
53
54
       \ensuremath{//} create the file for the filesystem
      int fd = open(fs_name, O_RDWR | O_CREAT | O_TRUNC, 0644);
57
      if (fd == -1) {
P_ERRNO = P_EOPEN;
58
59
60
         return -1;
61
      // extend the file to the required size
if (ftruncate(fd, filesystem_size) == -1) {
   P_ERRNO = P_EFUNC;
63
64
65
         close(fd);
66
         return -1;
68
69
70
      // allocate the FAT
      uint16_t* temp_fat = (uint16_t*)calloc(fat_entries, sizeof(uint16_t));
71
      if (!temp_fat) {
  P_ERRNO = P_EMALLOC;
72
73
74
         close(fd);
75
         return -1;
76
77
      // initialize FAT entries to their correct values
78
      temp_fat[0] = (num_blocks « 8) | blk_size;
temp_fat[1] = FAT_EOF;
for (int i = 2; i < fat_entries; i++) {</pre>
81
        temp_fat[i] = FAT_FREE;
82
8.3
84
     // write the FAT to the file
85
      if (write(fd, temp_fat, fat_size) != fat_size) {
```

```
P_ERRNO = P_EWRITE;
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) {
96
       P_ERRNO = P_ELSEEK;
       free(temp_fat);
free(root_dir);
97
98
99
       close(fd);
100
        return -1;
101
102
      if (write(fd, root_dir, actual_block_size) != actual_block_size) {
        P ERRNO = P EWRITE:
103
        free(temp_fat);
104
105
        free (root_dir);
106
        close(fd);
107
        return -1;
108
109
      // clean up
110
      free(temp_fat);
free(root_dir);
111
112
113
      close(fd);
114
      return 0;
115 }
```

4.1.1.7 mount()

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

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

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

Definition at line 120 of file fat_routines.c.

```
120
        // check if a filesystem is already mounted
121
        if (is_mounted) {
122
         P_ERRNO = P_EBUSY;
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) {
   P_ERRNO = P_ENOENT;
128
129
130
131
          return -1;
132
133
        // read the first two bytes to get size configuration
134
        uint16_t config;
135
        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
        // extract FAT region size information
       num_fat_blocks = (config » 8) & 0xFF; // MSB
int block_size_config = config & 0xFF; // LSB
int block_sizes[] = {256, 512, 1024, 2048, 4096};
block_size = block_sizes[block_size_config];
144
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);
154
          fs_fd = -1;
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
        close(fs_fd);
161
       fs_fd = -1;
162
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 (
     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 561 of file fat_routines.c.

```
561
562
                                                 char** args = (char**)arg;
 563
 564
                                                 // verify that the file system is mounted
 565
                                               if (!is_mounted) {
                                                          P_ERRNO = P_EFS_NOT_MOUNTED;
u_perror("mv");
 566
 567
 568
                                                              return NULL;
 569
 570
  571
                                                 // check if we have both source and destination arguments
 572
                                               if (args[1] == NULL || args[2] == NULL) {
                                                             P_ERRNO = P_EINVAL;
 573
                                                              u_perror("mv");
 574
 575
                                                              return NULL;
 576
 577
 578
                                               char* source = args[1];
 579
                                                 char* dest = args[2];
 580
                                                 // check if they're trying to rename to the same name
 581
 582
                                               if (strcmp(source, dest) == 0) {
 583
                                                             return NULL;
 584
 585
 586
                                                 // check if source file exists % \left( 1\right) =\left( 1\right) \left( 1\right) \left
                                              dir_entry_t source_entry;
int source_offset = find_file(source, &source_entry);
if (source_offset < 0) {</pre>
 587
 588
 589
                                                     u_perror("mv");
return NULL;
 590
 591
 592
                                              }
```

```
593
594
      // check if the destination file already exists
595
      dir_entry_t dest_entry;
      int dest_offset = find_file(dest, &dest_entry);
596
597
598
      // destination file exists
599
      if (dest_offset >= 0) {
600
        // check if the destination file is currently open by any process
601
        for (int i = 0; i < MAX_FDS; i++) {</pre>
          if (fd_table[i].in_use && strcmp(fd_table[i].filename, dest) == 0) {
602
            P_ERRNO = P_EBUSY;
u_perror("mv");
603
604
605
            return NULL;
606
607
608
        // if destination file exists, delete it
609
        if (mark_entry_as_deleted(&dest_entry, dest_offset) != 0) {
  u_perror("mv");
610
611
612
          return NULL;
613
614
615
616
      // rename file
617
      strncpy(source_entry.name, dest, sizeof(source_entry.name) - 1);
      source_entry.name[sizeof(source_entry.name) - 1] = '\0';
618
619
620
      // write the updated entry back to disk
      if (lseek(fs_fd, source_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
621
622
        u_perror("mv");
623
624
        return NULL;
625
626
627
      if (write(fs_fd, &source_entry, sizeof(source_entry)) !=
       sizeof(source_entry)) {
P_ERRNO = P_EWRITE;
u_perror("mv");
628
629
630
631
        return NULL;
632
633
     return NULL;
634
635 }
```

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 698 of file fat routines.c.

```
704
         u_perror("rm");
705
         return NULL;
706
707
      // check if we have any arguments
if (args[1] == NULL) {
708
709
       P_ERRNO = P_EINVAL;
710
711
         u_perror("rm");
712
         return NULL;
713
714
       // process each file argument
715
       for (int i = 1; args[i] != NULL; i++) {
   // find the file in the directory
716
717
718
         dir_entry_t entry;
719
720
         int entry_offset = find_file(args[i], &entry);
721
         if (entry_offset < 0) {
   // file doesn't exist</pre>
722
723
           P_ERRNO = P_ENOENT;
724
           u_perror("rm");
725
           continue;
726
727
728
         // check if file is currently open
729
         for (int j = 0; j < MAX_FDS; j++) {</pre>
730
               (fd_table[j].in_use && strcmp(fd_table[j].filename, args[i]) == 0) {
731
             P_ERRNO = P_EBUSY;
              u_perror("rm");
732
733
              continue;
734
735
         }
736
737
         \ensuremath{//} mark the directory entry as deleted
         if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
738
739
           u_perror("rm");
740
741
           continue;
742
743
         char deleted = 1; // mark as deleted
if (write(fs_fd, &deleted, sizeof(deleted)) != sizeof(deleted)) {
   P_ERRNO = P_EWRITE;
744
745
746
           u_perror("rm");
747
748
           continue;
749
750
         // free the FAT chain for this file
7.5.1
752
         uint16_t block = entry.firstBlock;
while (block != FAT_FREE && block != FAT_EOF) {
753
754
          uint16_t next_block = fat[block];
755
            fat[block] = FAT_FREE;
756
           block = next_block;
757
        }
      }
758
759
      return NULL;
761 }
```

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 501 of file fat_routines.c.

```
501 {
502 char** args = (char**)arg;
503
504 // verify that the file system is mounted
```

```
505
      if (!is_mounted) {
       P_ERRNO = P_EFS_NOT_MOUNTED;
u_perror("touch");
506
507
        return NULL;
508
509
510
511
      // check if we have any arguments
512
       if (args[1] == NULL) {
       P_ERRNO = P_EINVAL;
u_perror("touch");
513
514
        return NULL;
515
516
517
518
      // process each file argument
519
       for (int i = 1; args[i] != NULL; i++) {
        dir_entry_t entry;
int entry_offset = find_file(args[i], &entry);
520
521
522
523
         // file exists
524
         if (entry_offset >= 0) {
525
           entry.mtime = time(NULL);
526
           \ensuremath{//} write the updated entry back to the directory
52.7
           if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
  P_ERRNO = P_ELSEEK;
  u_perror("touch");
528
529
530
531
532
           if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
   P_ERRNO = P_EWRITE;
533
534
             u_perror("touch");
535
536
             continue;
537
538
         } else {
539
           // file doesn't exist, create a new empty file
540
           // check if the fat is full
if (P_ERRNO == P_EFULL) {
541
543
             u_perror("touch");
544
             return NULL;
545
546
           // add the file entry to root directory
547
           if (add_file_entry(args[i], 0, 0, TYPE_REGULAR, PERM_READ_WRITE) == -1) {
   u_perror("touch");
548
549
550
              continue;
551
552
        }
      }
553
554
      return NULL;
556 }
```

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
        if (munmap(fat, fat_size) == -1) {
  P_ERRNO = P_EMAP;
  return -1;
183
184
185
186
187
         fat = NULL;
188
       }
```

```
189
       // close fs_fd
if (fs_fd != -1) {
   if (close(fs_fd) == -1) {
     P_ERRNO = P_ECLOSE;
     return -1;
}
190
191
192
193
194
195
196
          fs\_fd = -1;
197
198
        // reset the other globals
199
        num_fat_blocks = 0;
200
201
       block_size = 0;
       fat_size = 0;
203
       is_mounted = false;
204 return 0;
205 }
```

4.1.2 Variable Documentation

4.1.2.1 current_running_pcb

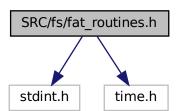
```
pcb_t* current_running_pcb [extern]
```

Definition at line 38 of file scheduler.c.

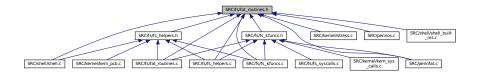
4.2 SRC/fs/fat_routines.h File Reference

```
#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 38 of file fat_routines.h.

4.2.1.2 F_READ

#define F_READ 0x01

Definition at line 36 of file fat_routines.h.

4.2.1.3 F_WRITE

#define F_WRITE 0x02

Definition at line 37 of file fat_routines.h.

4.2.1.4 FAT_EOF

#define FAT_EOF 0xFFFF

Definition at line 17 of file fat_routines.h.

4.2.1.5 **FAT_FREE**

#define FAT_FREE 0x0000

Definition at line 18 of file fat_routines.h.

4.2.1.6 PERM_EXEC

```
#define PERM_EXEC 4
```

Definition at line 30 of file fat_routines.h.

4.2.1.7 **PERM_NONE**

```
#define PERM_NONE 0
```

Definition at line 27 of file fat_routines.h.

4.2.1.8 PERM_READ

```
#define PERM_READ 2
```

Definition at line 29 of file fat_routines.h.

4.2.1.9 PERM_READ_EXEC

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

Definition at line 32 of file fat_routines.h.

4.2.1.10 PERM_READ_WRITE

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

Definition at line 31 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 33 of file fat_routines.h.

4.2.1.12 **PERM_WRITE**

```
#define PERM_WRITE 1
```

Definition at line 28 of file fat routines.h.

4.2.1.13 TYPE_DIRECTORY

```
#define TYPE_DIRECTORY 2
```

Definition at line 23 of file fat routines.h.

4.2.1.14 TYPE_REGULAR

```
#define TYPE_REGULAR 1
```

Definition at line 22 of file fat_routines.h.

4.2.1.15 TYPE_SYMLINK

```
#define TYPE_SYMLINK 4
```

Definition at line 24 of file fat_routines.h.

4.2.1.16 TYPE UNKNOWN

```
#define TYPE_UNKNOWN 0
```

Definition at line 21 of file fat_routines.h.

4.2.2 Function Documentation

4.2.2.1 cat()

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

Concatenates and displays files.

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

Usage formats:

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

Parameters

arg Arguments array (command line arguments)

Returns

284

u_perror("cat");

void pointer (unused)

```
Definition at line 214 of file fat_routines.c.
       char** args = (char**)arg;
215
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 we're
235
           // appending
           if ((strcmp(file_1, file_2) == 0) && is_append) {
236
            P_ERRNO = P_EREDIR;
u_perror("cat");
237
238
239
             return NULL;
240
241
           // edge case when input and output files names are the same but we're not // appending truncates the file \,
242
243
           if ((strcmp(file_1, file_2) == 0)) {
245
            return NULL;
246
247
           \ensuremath{//} 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);
252
             u_perror("cat");
253
             return NULL;
2.54
           if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
255
256
             k_close(in_fd);
             u_perror("cat");
258
             return NULL;
259
260
261
           char* buffer = (char*)malloc(block size);
           if (buffer == NULL) {
   P_ERRNO = P_EMALLOC;
262
263
264
             k_close(in_fd);
265
             u_perror("cat");
266
             return NULL;
2.67
268
269
           int bytes_read;
270
           ssize_t bytes_remaining = in_fd_size;
271
272
           while (bytes_remaining > 0) {
273
            ssize_t bytes_to_read =
                  bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
274
             bytes_read = k_read(in_fd, buffer, bytes_to_read);
275
276
277
             if (bytes_read <= 0) {</pre>
            break;
278
279
280
             if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
281
                free (buffer);
283
                k_close(in_fd);
```

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

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

```
459    k_close(in_fd);
460    free(buffer);
461  }
462
463    // close output file if not stdout
464    if (out_fd != STDOUT_FILENO) {
465        k_close(out_fd);
466    }
467
468    return NULL;
469 }
```

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 770 of file fat_routines.c.

```
782
       }
783
        // Find the file and get its current directory entry
784
785
       dir_entry_t dir_entry;
       int entry_offset = find_file(args[2], &dir_entry);
if (entry_offset < 0) {
   P_ERRNO = P_ENOENT;</pre>
786
787
788
789
          return NULL;
790
791
792
       // Calculate new permissions
       uint8_t new_perm = dir_entry.perm;
int i = 1; // Start after + or -
while (perm_str[i] != '\0') {
793
794
795
         switch (perm_str[i]) {
  case 'r':
796
797
               if (perm_str[0] == '+') {
  new_perm |= PERM_READ;
}
798
799
               } else {
800
               new_perm &= ~PERM_READ;
}
801
802
               break;
803
            case 'w':
  if (perm_str[0] == '+') {
   new_perm |= PERM_WRITE;
804
805
806
              new_perm &= ~PERM_WRITE;
}
808
809
            break; case 'x':
810
811
812
              if (perm_str[0] == '+') {
                 new_perm |= PERM_EXEC;
813
814
815
                 new_perm &= ~PERM_EXEC;
816
817
               break;
            default:
818
819
              P_ERRNO = P_EINVAL;
820
               return NULL;
821
822
         i++;
       }
823
824
       // Update the directory entry
825
       dir_entry.perm = new_perm;
827
       dir_entry.mtime = time(NULL);
828
       // Seek to the entry's position
if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
829
830
831
832
          return NULL;
833
834
       if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
P_ERRNO = P_EWRITE;
835
836
837
838
          return NULL;
839
840
841
       return NULL;
842 }
```

4.2.2.3 cmpctdir()

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

Compacts the root directory by removing all deleted entries.

Parameters

arg | Arguments array (command line arguments)

Returns

void pointer (unused)

Compacts the root directory by removing all deleted entries.

Definition at line 851 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 640 of file fat_routines.c.

```
// cp -h SOURCE DEST
if (strcmp(args[1], "-h") == 0) {
650
651
       if (args[2] == NULL || args[3] == NULL) {
   P_ERRNO = P_EINVAL;
   u_perror("cp");
652
653
654
655
           return NULL;
656
657
658
         if (copy_host_to_pennfat(args[2], args[3]) != 0) {
659
           u_perror("cp");
           return NULL;
660
661
         return NULL;
662
663
664
665
       // cp SOURCE -h DEST
       if (args[2] != NULL && strcmp(args[2], "-h") == 0) {
666
        if (args[2] := NOLL && SI
if (args[3] == NULL) {
   P_ERRNO = P_EINVAL;
667
668
669
           u_perror("cp");
670
           return NULL;
671
672
         if (copy_pennfat_to_host(args[1], args[3]) != 0) {
  u_perror("cp");
673
674
675
           return NULL;
676
677
         return NULL;
678
679
680
       // cp SOURCE DEST
       if ((args[1] != NULL && strcmp(args[1], "-h") != 0) && (args[2] != NULL && strcmp(args[2], "-h") != 0) && args[3] == NULL) {
681
682
683
         if (copy_source_to_dest(args[1], args[2]) != 0) {
684
          u_perror("cp");
685
           return NULL;
686
687
         return NULL;
688
689
      P_ERRNO = P_EUNKNOWN;
690
      u_perror("cp");
691
692
      return NULL;
693 }
```

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 476 of file fat_routines.c.

```
477
      // Note: we already check if fs is mounted in k_ls
478
      char** args = (char**)arg;
if (args[1] != NULL) {
479
480
       if (k_ls(args[1]) == -1) {
481
482
          u_perror("ls");
483
          return NULL;
484
485
     } else {
       if (k_ls(NULL) == -1) {
486
487
          u_perror("ls");
          return NULL;
488
489
490 }
491
492
     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
36
    if (num_blocks < 1 || num_blocks > 32) {
   P_ERRNO = P_EINVAL;
37
      return -1;
38
39
40
    if (blk_size < 0 || blk_size > 4) {
41
       P_ERRNO = P_EINVAL;
      return -1;
42
43
44
    // determine the file system size
45
     int block_sizes[] = {256, 512, 1024, 2048, 4096};
     int actual_block_size = block_sizes[blk_size];
48
    int fat_size = num_blocks * actual_block_size;
    int fat_entries = fat_size / 2;
49
50
    int num_data_blocks =
         (num\_blocks == 32)
             ? fat_entries - 2
              : fat_entries - 1; // note: first entry is reserved for metadata!
    size_t filesystem_size = fat_size + (actual_block_size * num_data_blocks);
54
5.5
56
    // create the file for the filesystem
57
    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;
6.3
64
65
         close(fd);
67
         return -1;
68
69
70
      // allocate the FAT
      uint16_t* temp_fat = (uint16_t*)calloc(fat_entries, sizeof(uint16_t));
71
      if (!temp_fat) {
   P_ERRNO = P_EMALLOC;
72
73
74
75
         close(fd);
         return -1:
      }
76
       // initialize FAT entries to their correct values
      // initialize FAT entries to their correct
temp_fat[0] = (num_blocks « 8) | blk_size;
temp_fat[1] = FAT_EOF;
for (int i = 2; i < fat_entries; i++) {
  temp_fat[i] = FAT_FREE;</pre>
79
80
81
82
83
84
85
      \ensuremath{//} write the FAT to the file
86
      if (write(fd, temp_fat, fat_size) != fat_size) {
87
         P_ERRNO = P_EWRITE;
88
         free(temp_fat);
89
         close(fd);
90
         return -1;
91
92
      // initialize the root directory + write to memory
uint8_t* root_dir = (uint8_t*)calloc(actual_block_size, 1);
if (lseek(fd, fat_size, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
93
94
95
96
         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) {
103
         P_ERRNO = P_EWRITE;
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);
113
       close(fd);
114
        return 0;
```

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 mou
--

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.

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

4.2.2.8 mv()

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 561 of file fat_routines.c.
```

```
562
      char** args = (char**)arg;
563
      \ensuremath{//} verify that the file system is mounted
564
      if (!is_mounted) {
565
       P_ERRNO = P_EFS_NOT_MOUNTED;
566
567
        u_perror("mv");
568
        return NULL;
569
570
      // check if we have both source and destination arguments
571
      if (args[1] == NULL || args[2] == NULL) {
573
       P_ERRNO = P_EINVAL;
574
        u_perror("mv");
575
        return NULL;
576
577
578
      char* source = args[1];
579
      char* dest = args[2];
580
581
      // check if they're trying to rename to the same name
582
     if (strcmp(source, dest) == 0) {
583
584
585
586
      // check if source file exists
587
      dir_entry_t source_entry;
      int source_offset = find_file(source, &source_entry);
588
      if (source_offset < 0) {
  u_perror("mv");</pre>
589
590
591
        return NULL;
592
593
594
      // check if the destination file already exists
      dir_entry_t dest_entry;
595
      int dest_offset = find_file(dest, &dest_entry);
596
597
598
      // destination file exists
599
      if (dest_offset >= 0) {
600
        // check if the destination file is currently open by any process
        for (int i = 0; i < MAX_FDS; i++) {
  if (fd_table[i].in_use && strcmp(fd_table[i].filename, dest) == 0) {</pre>
601
602
            P_ERRNO = P_EBUSY;
603
            u_perror("mv");
604
605
             return NULL;
606
          }
607
608
        // if destination file exists, delete it
609
        if (mark_entry_as_deleted(&dest_entry, dest_offset) != 0) {
611
          u_perror("mv");
612
           return NULL;
613
        }
614
615
616
      // rename file
      strncpy(source_entry.name, dest, sizeof(source_entry.name) - 1);
618
      source_entry.name[sizeof(source_entry.name) - 1] = '\0';
619
62.0
      // write the updated entry back to disk
if (lseek(fs_fd, source_offset, SEEK_SET) == -1) {
621
       P_ERRNO = P_ELSEEK;
u_perror("mv");
622
623
624
        return NULL;
62.5
626
      if (write(fs_fd, &source_entry, sizeof(source_entry)) !=
627
        sizeof(source_entry)) {
P_ERRNO = P_EWRITE;
u_perror("mv");
628
630
631
        return NULL;
```

```
632 }
633 
634 return NULL;
```

4.2.2.9 rm()

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

Removes files.

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

Parameters

arg | Arguments array (command line arguments)

Returns

void pointer (unused)

Definition at line 698 of file fat_routines.c.

```
char** args = (char**)arg;
699
700
701
         // verify that the file system is mounted
        if (!is_mounted) {
   P_ERRNO = P_EFS_NOT_MOUNTED;
702
703
           u_perror("rm");
704
705
           return NULL;
706
707
708
        // check if we have any arguments
        if (args[1] == NULL) {
  P_ERRNO = P_EINVAL;
  u_perror("rm");
709
710
711
712
           return NULL;
713
714
        // process each file argument
715
        // proceed claim the displanming
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);
716
717
718
719
720
           if (entry_offset < 0) {
   // file doesn't exist
   P_ERRNO = P_ENOENT;</pre>
721
722
723
724
              u_perror("rm");
725
              continue;
726
727
728
           // check if file is currently open
for (int j = 0; j < MAX_FDS; j++) {
   if (fd_table[j].in_use && strcmp(fd_table[j].filename, args[i]) == 0) {</pre>
729
730
731
                 P_ERRNO = P_EBUSY;
732
                 u_perror("rm");
733
                 continue;
734
             }
735
736
737
           // mark the directory entry as deleted
738
           if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
            P_ERRNO = P_ELSEEK;
u_perror("rm");
739
740
741
              continue;
742
```

```
char deleted = 1;  // mark as deleted
if (write(fs_fd, &deleted, sizeof(deleted)) != sizeof(deleted)) {
744
745
           P_ERRNO = P_EWRITE;
u_perror("rm");
746
747
748
             continue:
749
750
751
          \ensuremath{//} free the FAT chain for this file
          uint16_t block = entry.firstBlock;
while (block != FAT_FREE && block != FAT_EOF) {
752
753
          uint16_t next_block = fat[block];
fat[block] = FAT_FREE;
754
755
756
             block = next_block;
757
758 }
759
760
       return NULL;
```

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 501 of file fat_routines.c.

```
502
      char** args = (char**)arg;
503
504
      // verify that the file system is mounted
505
      if (!is_mounted) {
       P_ERRNO = P_EFS_NOT_MOUNTED;
u_perror("touch");
506
507
508
        return NULL;
509
510
      // check if we have any arguments
511
      if (args[1] == NULL) {
512
       P_ERRNO = P_EINVAL;
u_perror("touch");
513
514
515
        return NULL;
516
517
      // process each file argument
      for (int i = 1; args[i] != NULL; i++) {
       dir_entry_t entry;
        int entry_offset = find_file(args[i], &entry);
521
522
523
        // file exists
524
        if (entry_offset >= 0) {
          entry.mtime = time(NULL);
```

```
526
527
            // write the updated entry back to the directory
           if (lseek(fs_fd, entry_offset, SEEK_SET) == -1)
   P_ERRNO = P_ELSEEK;
   u_perror("touch");
528
529
530
531
             continue:
532
533
           if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
            P_ERRNO = P_EWRITE;
u_perror("touch");
534
535
536
             continue;
537
538
        } else {
539
           // file doesn't exist, create a new empty file
540
541
           // check if the fat is full
           if (P_ERRNO == P_EFULL) {
  u_perror("touch");
542
543
544
             return NULL;
545
546
547
           // add the file entry to root directory
           if (add_file_entry(args[i], 0, 0, TYPE_REGULAR, PERM_READ_WRITE) == -1) {
   u_perror("touch");
548
549
550
              continue;
551
552
      }
553
554
555
      return NULL;
556 }
```

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.

```
175
       // 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
182
       if (munmap(fat, fat_size) == -1) {
  P_ERRNO = P_EMAP;
  return -1;
183
184
185
186
187
         fat = NULL;
188
189
      // close fs_fd
if (fs_fd != -1) {
190
191
        if (close(fs_fd) == -1) {
192
193
          P_ERRNO = P_ECLOSE;
194
           return -1;
195
         fs_fd = -1:
196
197
198
199
      // reset the other globals
200
       num_fat_blocks = 0;
      block_size = 0;
fat_size = 0;
201
202
203
      is_mounted = false;
204
      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 267 of file fs_helpers.c.
```

```
272
      if (!is_mounted) {
       P_ERRNO = P_EFS_NOT_MOUNTED;
return -1;
273
274
275
276
277
      // check if file already exists
278
      dir_entry_t existing;
279
      if (find_file(filename, &existing) >= 0) {
      P_ERRNO = P_EEXIST;
280
       return -1;
281
282
283
284
      // start with root directory block (block 1)
285
      uint16_t current_block = 1;
     int offset = 0;
dir_entry_t dir_entry;
286
287
288
289
      while (1) {
290
       // position at the start of current block of the root directory
291
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
        -1) {
P_ERRNO = P_ELSEEK;
292
293
294
          return -1;
295
296
297
        // reset offset for new block
298
        offset = 0;
299
300
        // search current block for free slot
        while (offset < block_size) {</pre>
301
302
         if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
303
            P_ERRNO = P_EREAD;
304
            return -1;
305
306
307
          // found a free slot
          if (dir_entry.name[0] == 0 || dir_entry.name[0] == 1) {
```

```
// initialize the new entry
310
            memset(&dir_entry, 0, sizeof(dir_entry));
311
            strncpy(dir_entry.name, filename, 31);
312
            dir_entry.size = size;
            dir_entry.firstBlock = first_block;
313
            dir_entry.type = type;
dir_entry.perm = perm;
314
315
            dir_entry.mtime = time(NULL);
316
317
318
            // write the entry
            319
320
              P_ERRNO = P_ELSEEK;
321
322
              return -1;
323
324
            if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
325
              P_ERRNO = P_EWRITE;
              return -1;
326
327
328
329
            return offset;
330
331
          offset += sizeof(dir_entry);
332
333
334
335
        // current block is full, check if there's a next block
336
        if (fat[current_block] != FAT_EOF) {
337
         current_block = fat[current_block];
338
          continue;
339
340
341
        // allocate a new block for the root directory
342
        uint16_t new_block = allocate_block();
        if (new_block == 0) {
  P_ERRNO = P_EFULL;
343
344
          return -1;
345
346
347
348
        // chain the new block
        fat[current_block] = new_block;
fat[new_block] = FAT_EOF;
349
350
351
352
        // initialize new block
353
        uint8_t* zero_block = calloc(block_size, 1);
354
        if (!zero_block) {
355
         P_ERRNO = P_EINVAL;
356
          return -1;
357
358
359
        // write this new block to the file system
360
        if (lseek(fs_fd, fat_size + (new_block - 1) * block_size, SEEK_SET) == -1) {
361
          P_ERRNO = P_ELSEEK;
          free(zero_block);
362
363
          return -1:
364
365
        if (write(fs_fd, zero_block, block_size) != block_size) {
366
          P_ERRNO = P_EWRITE;
367
          free(zero_block);
368
          return -1;
369
370
371
        free (zero_block);
372
373
        // initialize the new entry
374
        memset(&dir_entry, 0, sizeof(dir_entry));
375
        strncpy(dir_entry.name, filename, 31);
376
        dir_entry.size = size;
        dir_entry.firstBlock = first_block;
377
378
        dir_entry.type = type;
379
        dir_entry.perm = perm;
380
        dir_entry.mtime = time(NULL);
381
        // write the new entry at the start of the new block in the file system
if (lseek(fs_fd, fat_size + (new_block - 1) * block_size, SEEK_SET) == -1) {
382
383
384
          P_ERRNO = P_ELSEEK;
385
          return -1;
386
387
        if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
388
          P ERRNO = P EWRITE;
          return -1;
389
390
391
392
        return 0;
393
     }
394 }
```

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 166 of file fs_helpers.c.

```
167
        for (int i = 2; i < fat_size / 2; i++) {</pre>
          if (fat[i] == FAT_FREE) {
  fat[i] = FAT_EOF;
  return i;
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
                return i;
179
      }
180
181
182
183
       return 0;
```

4.3.1.3 compact_directory()

```
int compact_directory ( )
```

Compacts a directory.

Compacts the root directory by removing all deleted entries.

Definition at line 699 of file fs_helpers.c.

```
699
700
      if (!is_mounted) {
701
       P_ERRNO = P_EFS_NOT_MOUNTED;
        return -1;
702
703
704
705
      // buffer for temp storage of a block
706
      uint8_t* dir_buffer = malloc(block_size);
      if (!dir_buffer) {
707
      P_ERRNO = P_EMALLOC;
return -1;
708
709
710
711
      // start at root directory
     uint16_t current_block = 1;
int dir_entries_count = 0;
713
714
715
     int deleted_entries_count = 0;
716
      // calculate number of entries and deleted entries in the root directory
718
      while (current_block != FAT_EOF) {
719
      if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
         -1) {
P_ERRNO = P_ELSEEK;
720
721
722
          free(dir_buffer);
723
          return -1;
724
```

```
726
        if (read(fs_fd, dir_buffer, block_size) != block_size) {
          P_ERRNO = P_EREAD;
727
          free(dir_buffer);
728
729
          return -1;
730
731
732
        // count entries and deleted entries in this block
733
        for (int offset = 0; offset < block_size; offset += sizeof(dir_entry_t)) {</pre>
734
          dir_entry_t* entry = (dir_entry_t*) (dir_buffer + offset);
735
          // check if we've reached the end of directory
736
          if (entry->name[0] == 0) {
737
738
            break;
739
740
741
          dir_entries_count++;
742
743
          // check if it's a deleted entry
744
          if (entry->name[0] == 1) {
745
            deleted_entries_count++;
746
747
       }
748
749
        // move onto next block, if there is one
750
        if (fat[current_block] != FAT_EOF)
751
          current_block = fat[current_block];
752
        } else {
753
          break;
754
        }
755
756
757
      // if no deleted entries, no compaction needed
758
      if (deleted_entries_count == 0) {
759
       free(dir_buffer);
760
        return 0;
761
762
763
      // allocate space for all valid entries
764
      dir_entry_t* all_entries = malloc(dir_entries_count * sizeof(dir_entry_t));
765
      if (!all_entries) {
       P_ERRNO = P_EMALLOC;
free(dir_buffer);
766
767
768
        return -1;
769
770
771
      // read all entries into the buffer, skipping deleted ones
772
      current block = 1;
773
      int valid_entry_idx = 0;
774
775
      while (current_block != FAT_EOF) {
776
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
777
             -1) {
          P_ERRNO = P_ELSEEK;
778
779
          free(dir_buffer);
780
          free(all entries);
781
          return -1;
782
783
        if (read(fs_fd, dir_buffer, block_size) != block_size) {
   P_ERRNO = P_EREAD;
784
785
          free (dir_buffer);
786
787
          free(all_entries);
788
          return -1;
789
790
791
        \ensuremath{//} process entries in this block
        for (int offset = 0; offset < block_size; offset += sizeof(dir_entry_t)) {</pre>
792
          dir_entry_t* entry = (dir_entry_t*) (dir_buffer + offset);
793
794
795
          // check if we've reached the end of directory
796
          if (entry->name[0] == 0) {
797
            break;
798
799
          // skip deleted entries
800
801
          if (entry->name[0] == 1) {
802
            continue;
803
804
805
          // copy valid entry to our array
          memcpy(&all_entries[valid_entry_idx++], entry, sizeof(dir_entry_t));
806
807
808
        // move to the next block
if (fat[current_block] != FAT_EOF) {
809
810
          current_block = fat[current_block];
811
```

```
812
       } else {
813
          break;
814
        }
815
      }
816
      // rewrite the directory with only valid entries
817
      current_block = 1;
818
819
      int entries_per_block = block_size / sizeof(dir_entry_t);
820
      int blocks_needed =
           (valid_entry_idx + entries_per_block - 1) / entries_per_block;
821
822
      \ensuremath{//} clean up any excess directory blocks in the FAT chain
823
      uint16_t next_block = fat[current_block];
if (blocks_needed == 1) {
824
825
826
        // only need one block, free all others
827
        while (next_block != FAT_EOF) {
828
          uint16_t temp = fat[next_block];
           fat[next_block] = FAT_FREE;
829
830
          next_block = temp;
831
832
        fat[current_block] = FAT_EOF;
833
834
        // navigate through needed blocks
835
        int block count = 1;
836
        uint16_t prev_block = current_block;
837
838
        while (block_count < blocks_needed) {</pre>
839
          if (next_block == FAT_EOF) {
840
             // need to allocate a new block
841
             uint16_t new_block = allocate_block();
             if (new_block == 0) {
842
843
               P_ERRNO = P_EFULL;
844
               free(dir_buffer);
845
               free(all_entries);
846
              return -1;
847
             fat[prev block] = new block;
848
            next_block = new_block;
850
851
          prev_block = next_block;
next_block = fat[next_block];
852
853
854
          block_count++;
855
856
857
         // free any excess blocks
858
        fat[prev_block] = FAT_EOF;
859
        while (next_block != FAT_EOF) {
860
          uint16_t temp = fat[next_block];
           fat[next_block] = FAT_FREE;
861
862
          next_block = temp;
863
864
865
      \ensuremath{//} write the valid entries back to the directory blocks
866
      current_block = 1;
867
      int entries_written = 0;
868
869
870
      while (entries_written < valid_entry_idx) {</pre>
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
871
872
             -1) {
          P_ERRNO = P_ELSEEK;
873
           free(dir_buffer);
875
           free(all_entries);
876
           return -1;
877
878
879
        memset (dir buffer, 0, block size);
880
881
         // copy entries to the buffer
882
         int entries_in_this_block = 0;
883
        while (entries_written < valid_entry_idx &&</pre>
884
                entries_in_this_block < entries_per_block) {</pre>
885
          \label{lem:memcpy} \verb| (dir_buffer + (entries_in_this_block * sizeof(dir_entry_t)) |,
                  &all_entries[entries_written], sizeof(dir_entry_t));
886
           entries_written++;
887
888
          entries_in_this_block++;
889
890
        // write the buffer to the file system
891
        f (write(fs_fd, dir_buffer, block_size) != block_size) {
   P_ERRNO = P_EINVAL;
892
893
894
           free(dir_buffer);
895
          free(all_entries);
896
          return -1;
897
898
```

```
899    // move to the next block if needed
900    if (entries_written < valid_entry_idx) {
901         current_block = fat[current_block];
902    }
903    }
904
905    free(dir_buffer);
906    free(all_entries);
907    return 0;
908 }</pre>
```

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 438 of file fs_helpers.c.

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

```
493
        if (bytes_read <= 0) {</pre>
494
          break;
495
496
        // write to pennfat_fd using k\_write
497
        if (k_write(pennfat_fd, (const char*)buffer, bytes_read) != bytes_read) {
498
499
         free(buffer);
500
          k_close(pennfat_fd);
501
         close(host_fd);
502
          return -1;
       }
503
504
505
       bytes_remaining -= bytes_read;
506
507
508
      // check for read error
      if (bytes_read < 0) {
  P_ERRNO = P_EREAD;</pre>
509
510
511
        free (buffer);
512
        k_close(pennfat_fd);
513
       close(host_fd);
514
        return -1;
515
516
517
      // otherwise, cleanup and return success
518
     free (buffer);
519
      k_close(pennfat_fd);
520 close(host_fd);
521
      return 0;
522 }
```

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 527 of file fs_helpers.c.

```
528
                                                           {
529
      if (!is_mounted) {
530
       P_ERRNO = P_EFS_NOT_MOUNTED;
531
       return -1;
532
533
534
     // open the PennFAT file
      int pennfat_fd = k_open(pennfat_filename, F_READ);
535
     if (pennfat_fd < 0) {</pre>
537
       return -1;
538
539
540
     // get the pennfat file size
541
      off_t pennfat_file_size_in_bytes = k_lseek(pennfat_fd, 0, SEEK_END);
      if (pennfat_file_size_in_bytes == -1) {
542
543
      k_close(pennfat_fd);
544
       return -1;
545
546
547
      // go back to beginning of file for reading
      if (k_lseek(pennfat_fd, 0, SEEK_SET) == -1) {
548
      k_close (pennfat_fd);
549
550
551
552
553
     // open the host file
     int host_fd = open(host_filename, O_WRONLY | O_CREAT | O_TRUNC, 0644); if (host_fd == -1) {
554
555
556
       P_ERRNO = P_EOPEN;
557
       k_close(pennfat_fd);
558
       return -1;
559
560
     // allocate buffer for data transfer
```

```
562
      char* buffer = (char*)malloc(block_size);
563
      if (!buffer) {
        P_ERRNO = P_EMALLOC;
564
565
        k_close(pennfat_fd);
566
       close(host_fd);
567
        return -1:
568
569
570
      uint32_t bytes_remaining = pennfat_file_size_in_bytes;
571
      ssize_t bytes_read;
572
573
      // read from PennFAT file and write to host file
574
      while (bytes_remaining > 0) {
575
       // ensure bytes to read never exceeds the block size
576
        ssize_t bytes_to_read =
577
578
            bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
        bytes_read = k_read(pennfat_fd, buffer, bytes_to_read);
579
580
        if (bytes_read <= 0) {</pre>
581
         break;
582
583
584
        if (write(host_fd, buffer, bytes_read) != bytes_read) {
585
         P ERRNO = P EINVAL;
586
          free (buffer);
587
          close(host_fd);
588
          k_close(pennfat_fd);
589
          return -1;
590
591
592
        bytes_remaining -= bytes_read;
593
594
595
      // check for read error
      if (bytes_read < 0) {
  P_ERRNO = P_EREAD;</pre>
596
597
598
        free (buffer);
599
        close(host_fd);
600
        k_close(pennfat_fd);
601
602
603
     // otherwise, cleanup and return success
604
605
     free (buffer);
      close(host_fd);
607
      k_close(pennfat_fd);
608 return 0;
609 }
```

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 614 of file fs_helpers.c.

```
615
616
      if (!is mounted) {
       P_ERRNO = P_EFS_NOT_MOUNTED;
617
618
        return -1;
619
620
621
      // open the source file
      int source_fd = k_open(source_filename, F_READ);
622
      if (source_fd < 0) {</pre>
623
       return -1;
624
625
626
     \ensuremath{//} get the source file size
62.7
628
     off_t source_file_size_in_bytes = k_lseek(source_fd, 0, SEEK_END);
629
     if (source_file_size_in_bytes == -1) {
       k_close(source_fd);
```

```
631
        return -1;
632
633
      \ensuremath{//} move to the beginning of the source file for reading
634
635
      if (k_lseek(source_fd, 0, SEEK_SET) < 0) {</pre>
      k_close (source_fd);
636
637
        return -1;
638
639
      // open the destination file
int dest_fd = k_open(dest_filename, F_WRITE);
if (dest_fd < 0) {</pre>
640
641
642
       k_close(source_fd);
return -1;
643
644
645
646
      \ensuremath{//} read from source to destination
647
648
      char* buffer = (char*)malloc(block_size);
649
      if (!buffer) {
       P_ERRNO = P_EMALLOC;
650
651
        k_close(source_fd);
652
        k_close(dest_fd);
653
        return -1;
654
655
656
      uint32_t bytes_remaining = source_file_size_in_bytes;
657
      ssize_t bytes_read;
658
659
      while (bytes_remaining > 0) {
      // make sure the bytes to read doesn't exceed block size
ssize_t bytes_to_read =
660
661
662
            bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
663
        bytes_read = k_read(source_fd, buffer, bytes_to_read);
664
665
        if (bytes_read <= 0) {</pre>
666
          break;
        }
667
668
669
        if (k_write(dest_fd, buffer, bytes_read) != bytes_read) {
670
         free (buffer);
671
          k_close(source_fd);
672
          k_close(dest_fd);
673
          return -1;
674
        }
675
676
677
      // check for read error
678
      if (bytes_read < 0) {</pre>
679
        free (buffer);
        k_close(source_fd);
680
681
        k_close(dest_fd);
682
683
684
      // otherwise, cleanup and return success
685
686
      free (buffer);
      k_close(source_fd);
688
      k_close(dest_fd);
689
      return 0;
690 }
```

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.
```

```
112
113
      if (!fd_table[fd].in_use) {
114
        P_ERRNO = P_EBADF;
       return -1;
115
116
117
118
      fd_table[fd].ref_count--;
119
      if (fd_table[fd].ref_count == 0) {
120
       fd_table[fd].in_use = 0;
121
        memset(fd_table[fd].filename, 0, sizeof(fd_table[fd].filename));
        fd_table[fd].size = 0;
fd_table[fd].first_block = 0;
122
123
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 191 of file fs helpers.c.

```
191
192
      if (!is_mounted) {
193
       P_ERRNO = P_EFS_NOT_MOUNTED;
194
        return -1;
195
196
      // Start with root directory block (block 1)
197
      uint16_t current_block = 1;
198
199
      int offset_in_block = 0;
200
      int absolute_offset = 0;
201
      dir_entry_t dir_entry;
202
203
      while (1) {
204
        // Position at the start of current block
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
205
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
        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
if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
229
230
231
           offset_in_block += sizeof(dir_entry);
232
            absolute_offset += sizeof(dir_entry);
233
            continue;
234
235
236
          // check if we found the file
          if (strcmp(dir_entry.name, filename) == 0) {
```

```
238
           if (entry) {
239
            memcpy(entry, &dir_entry, sizeof(dir_entry));
240
           return absolute_offset; // return the absolute file offset
241
2.42
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
248
249
       // block
       if (fat[current_block] != FAT_EOF) {
250
251
        current_block = fat[current_block];
252
253
254
255
       // no more blocks to search
256
       break;
257
258
     // file not found
259
260 P_ERRNO = P_ENOENT;
2.61
     return -1;
262 }
```

4.3.1.9 get_free_fd()

Gets a free file descriptor.

Finds the first available file descriptor in the table.

```
Definition at line 77 of file fs_helpers.c.
```

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

4.3.1.10 has_executable_permission()

```
int has_executable_permission (  \qquad \qquad \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
      // check if fs is mounted
134
135
     if (!is_mounted) {
136
      P_ERRNO = P_EFS_NOT_MOUNTED;
137
       return -1;
138
139
140
     // validate fd argument
141
     if (fd < 0 || fd >= MAX_FDS) {
      P_ERRNO = P_EINVAL;
```

```
143     return -1;
144   }
145
146     // determine whether the file exists
147     dir_entry_t entry;
148     int entry_offset = find_file(fd_table[fd].filename, &entry);
149     if (entry_offset < 0) {
        return -1;
151     }
152
153     // if it exists, get its permission
154     if (entry.perm & PERM_EXEC) {
        return 1;
156     }
157
158     return 0;
159 }</pre>
```

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.

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
44 // STDIN (fd 0)
```

```
45
     fd_table[0].in_use = 1;
     fd_table[0].ref_count = 1;
     strncpy(fd_table[0].filename, "<stdin>", 31);
47
48
    fd_table[0].mode = F_READ;
49
     // STDOUT (fd 1)
50
51
     fd_table[1].in_use = 1;
     strncpy(fd_table[1].filename, "<stdout>", 31);
53
     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);
fd_table[2].mode = F_WRITE; // write-only
59
60
    fd_table[2].ref_count = 1;
61
     // other file descriptors (fd 3 and above) for (int i = 3; i < MAX\_FDS; i++) {
62
63
      fd_table[i].in_use = 0;
       fd_table[i].ref_count = 0;
66
       memset(fd_table[i].filename, 0, sizeof(fd_table[i].filename));
       fd_table[i].size = 0;
fd_table[i].first_block = 0;
67
68
69
       fd_table[i].position = 0;
       fd_table[i].mode = 0;
71
72 }
```

4.3.1.13 mark entry as deleted()

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 399 of file fs helpers.c.
```

```
399
400
      if (!is_mounted || entry == NULL || absolute_offset < 0) {</pre>
401
       P_ERRNO = P_EINVAL;
402
        return -1;
403
404
405
      // free the blocks
406
      uint16_t current_block = entry->firstBlock;
407
      while (current_block != FAT_FREE && current_block != FAT_EOF) {
        uint16_t next_block = fat[current_block];
fat[current_block] = FAT_FREE;
408
409
410
        current_block = next_block;
411
412
413
      // mark the entry as deleted in the root directory
414
      dir_entry_t deleted_entry = *entry;
415
      deleted_entry.name[0] = 1;
      if (lseek(fs_fd, absolute_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
416
417
418
        return -1;
419
420
      if (write(fs_fd, &deleted_entry, sizeof(deleted_entry)) !=
421
          sizeof(deleted_entry)) {
        P_ERRNO = P_EINVAL;
422
        return -1;
423
424
425
426
      // mark the passed entry as deleted
427
      entry->name[0] = 1;
428
      return 0;
429 }
```

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.

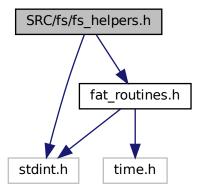
4.3.2.8 num_fat_blocks

int num_fat_blocks = 0

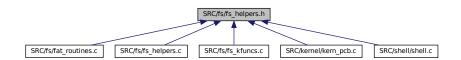
Definition at line 29 of file fs_helpers.c.

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 267 of file fs_helpers.c.
```

```
272
      if (!is mounted) {
273
        P_ERRNO = P_EFS_NOT_MOUNTED;
        return -1;
275
276
277
      // check if file already exists
      dir_entry_t existing;
if (find_file(filename, &existing) >= 0) {
278
279
       P_ERRNO = P_EEXIST;
280
281
        return -1;
282
283
      // start with root directory block (block 1)
uint16_t current_block = 1;
284
285
286
      int offset = 0;
      dir_entry_t dir_entry;
288
289
      while (1) {
        // 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) ==
291
292
             -1) {
           P_ERRNO = P_ELSEEK;
293
          return -1;
294
295
296
297
        // reset offset for new block
298
        offset = 0;
299
300
         // search current block for free slot
301
         while (offset < block_size) {</pre>
302
           if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
             P_ERRNO = P_EREAD;
303
304
             return -1;
305
306
307
           // found a free slot
308
           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
311
312
313
            dir_entry.firstBlock = first_block;
314
            dir_entry.type = type;
             dir_entry.perm = perm;
315
316
            dir_entry.mtime = time(NULL);
317
             // write the entry
318
319
             if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + offset,
               SEEK_SET) == -1) {
P_ERRNO = P_ELSEEK;
320
321
               return -1;
322
323
324
             if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
325
326
               return -1;
327
328
329
             return offset;
330
```

```
332
           offset += sizeof(dir_entry);
333
334
         // current block is full, check if there's a next block
if (fat[current_block] != FAT_EOF) {
335
336
          current_block = fat[current_block];
337
338
           continue;
339
340
         // allocate a new block for the root directory
uint16_t new_block = allocate_block();
if (new_block == 0) {
341
342
343
           P_ERRNO = P_EFULL;
344
345
           return -1;
346
347
         // chain the new block
348
         fat[current_block] = new_block;
fat[new_block] = FAT_EOF;
349
350
351
352
          // initialize new block
353
         uint8_t* zero_block = calloc(block_size, 1);
         if (!zero_block) {
  P_ERRNO = P_EINVAL;
354
355
356
           return -1;
357
358
359
         // 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;
360
361
362
           free (zero_block);
363
           return -1;
364
365
         if (write(fs_fd, zero_block, block_size) != block_size) {
366
           P_ERRNO = P_EWRITE;
           free(zero_block);
367
368
           return -1;
369
370
371
         free(zero_block);
372
373
         // initialize the new entry
         memset(&dir_entry, 0, sizeof(dir_entry));
strncpy(dir_entry.name, filename, 31);
dir_entry.size = size;
374
375
376
377
         dir_entry.firstBlock = first_block;
         dir_entry.type = type;
dir_entry.perm = perm;
378
379
         dir_entry.mtime = time(NULL);
380
381
         // write the new entry at the start of the new block in the file system
382
383
         if (lseek(fs_fd, fat_size + (new_block - 1) * block_size, SEEK_SET) == -1) {
384
           P_ERRNO = P_ELSEEK;
385
           return -1;
386
387
         if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
388
           P_ERRNO = P_EWRITE;
389
           return -1;
390
391
392
         return 0;
393
      }
394 }
```

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 166 of file fs_helpers.c.

```
166
167
          for (int i = 2; i < fat_size / 2; i++) {
   if (fat[i] == FAT_FREE) {
    fat[i] = FAT_EOF;</pre>
168
169
170
                return i;
171
172
173
         if (compact_directory() == 0) {
   for (int i = 2; i < fat_size / 2; i++) {
     if (fat[i] == FAT_FREE) {</pre>
174
176
                  fat[i] = FAT_EOF;
177
178
                   return i;
179
            }
180
181 }
183
        return 0;
184 }
```

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 699 of file fs_helpers.c.

```
699
700
      if (!is_mounted) {
701
      P_ERRNO = P_EFS_NOT_MOUNTED;
702
       return -1;
703
704
     // buffer for temp storage of a block
705
706
     uint8_t* dir_buffer = malloc(block_size);
707
     if (!dir_buffer) {
      P_ERRNO = P_EMALLOC;
708
709
       return -1;
710
711
      // start at root directory
712
713
     uint16_t current_block = 1;
714
      int dir_entries_count = 0;
715
     int deleted_entries_count = 0;
716
     // calculate number of entries and deleted entries in the root directory while (current_block != FAT_EOF) {
717
718
      if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
719
720
            -1) {
721
         P_ERRNO = P_ELSEEK;
722
         free(dir_buffer);
723
          return -1;
724
725
        if (read(fs_fd, dir_buffer, block_size) != block_size) {
```

```
P_ERRNO = P_EREAD;
728
                                  free(dir_buffer);
729
                                  return -1;
                           }
730
731
                            // count entries and deleted entries in this block
732
                            for (int offset = 0; offset < block_size; offset += sizeof(dir_entry_t)) {</pre>
733
734
                                  dir_entry_t* entry = (dir_entry_t*) (dir_buffer + offset);
735
                                  // check if we've reached the end of directory if (entry->name[0] == 0) {
736
737
738
                                       break;
739
740
741
                                  dir_entries_count++;
742
                                   // check if it's a deleted entry % \left( 1\right) =\left( 1\right) \left( 1\right)
743
                                  if (entry->name[0] == 1) {
744
745
                                       deleted_entries_count++;
746
                                  }
747
748
                           // move onto next block, if there is one
if (fat[current_block] != FAT_EOF) {
   current_block = fat[current_block];
749
750
751
                            } else {
753
754
                          }
755
756
                    // if no deleted entries, no compaction needed
if (deleted_entries_count == 0) {
757
 758
759
                        free(dir_buffer);
760
                           return 0;
761
762
                    // allocate space for all valid entries
763
                    dir_entry_t* all_entries = malloc(dir_entries_count * sizeof(dir_entry_t));
764
                    if (!all_entries) {
  P_ERRNO = P_EMALLOC;
765
766
767
                          free(dir_buffer);
768
                          return -1;
769
770
771
                    // read all entries into the buffer, skipping deleted ones
772
                     current_block = 1;
773
                    int valid_entry_idx = 0;
774
775
                    while (current_block != FAT_EOF) {
776
                        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
777
                                           -1) {
778
                                P_ERRNO = P_ELSEEK;
779
                                  free(dir_buffer);
780
                                 free(all_entries);
781
                                  return -1;
782
783
784
                           if (read(fs_fd, dir_buffer, block_size) != block_size) {
                              P_ERRNO = P_EREAD;
785
786
                                  free(dir_buffer);
787
                                  free(all entries);
788
                                  return -1;
789
790
791
                             // process entries in this block
792
                            for (int offset = 0; offset < block_size; offset += sizeof(dir_entry_t)) {</pre>
793
                                  dir_entry_t* entry = (dir_entry_t*) (dir_buffer + offset);
794
795
                                   // check if we've reached the end of directory
796
                                   if (entry->name[0] == 0) {
797
                                       break;
798
799
                                   // skip deleted entries
800
801
                                   if (entry->name[0] == 1) {
802
                                       continue;
803
804
805
                                   // copy valid entry to our array
                                 memcpy(&all_entries[valid_entry_idx++], entry, sizeof(dir_entry_t));
806
807
808
809
                             // move to the next block
810
                             if (fat[current_block] != FAT_EOF) {
811
                                current_block = fat[current_block];
812
                            } else {
813
                                  break:
```

```
814
        }
815
816
817
      // rewrite the directory with only valid entries
818
      current block = 1;
      int entries_per_block = block_size / sizeof(dir_entry_t);
819
      int blocks_needed =
821
           (valid_entry_idx + entries_per_block - 1) / entries_per_block;
822
823
      // clean up any excess directory blocks in the FAT chain \,
      uint16_t next_block = fat[current_block];
if (blocks_needed == 1) {
824
825
        // only need one block, free all others
while (next_block != FAT_EOF) {
826
827
828
          uint16_t temp = fat[next_block];
           fat[next_block] = FAT_FREE;
829
830
          next_block = temp;
831
832
        fat[current_block] = FAT_EOF;
833
      } else {
834
         // navigate through needed blocks
835
        int block_count = 1;
        uint16_t prev_block = current_block;
836
837
838
        while (block_count < blocks_needed) {</pre>
          if (next_block == FAT_EOF) {
839
840
             // need to allocate a new block
841
             uint16_t new_block = allocate_block();
             if (new_block == 0) {
  P_ERRNO = P_EFULL;
842
843
               free (dir_buffer);
844
845
               free(all_entries);
846
               return -1;
847
848
             fat[prev_block] = new_block;
849
            next_block = new_block;
850
851
          prev_block = next_block;
next_block = fat[next_block];
852
853
854
          block_count++;
855
856
857
        // free any excess blocks
        fat[prev_block] = FAT_EOF;
859
             e (next_block != FAT_EOF) {
860
          uint16_t temp = fat[next_block];
           fat[next_block] = FAT_FREE;
861
862
          next_block = temp;
863
864
      }
865
866
      // write the valid entries back to the directory blocks
867
      current_block = 1;
      int entries_written = 0;
868
869
870
      while (entries_written < valid_entry_idx) {</pre>
871
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
             -1) {
872
          P ERRNO = P_ELSEEK;
873
874
          free(dir_buffer);
875
          free(all entries);
876
          return -1;
877
878
879
        memset(dir_buffer, 0, block_size);
880
881
        // copy entries to the buffer
        int entries_in_this_block = 0;
882
883
        while (entries_written < valid_entry_idx &&</pre>
884
                entries_in_this_block < entries_per_block) {</pre>
          memcpy(dir_buffer + (entries_in_this_block * sizeof(dir_entry_t)),
885
886
                  &all_entries[entries_written], sizeof(dir_entry_t));
          entries_written++;
887
888
          entries_in_this_block++;
889
890
891
         // write the buffer to the file system
        if (write(fs_fd, dir_buffer, block_size) != block_size) {
   P_ERRNO = P_EINVAL;
892
893
894
          free(dir_buffer);
895
           free(all_entries);
          return -\overline{1};
896
897
898
         // move to the next block if needed
899
        if (entries_written < valid_entry_idx) {</pre>
900
```

```
901          current_block = fat[current_block];
902     }
903    }
904
905     free(dir_buffer);
906     free(all_entries);
907     return 0;
908 }
```

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 438 of file fs_helpers.c.

```
440
       if (!is_mounted) {
        P_ERRNO = P_EFS_NOT_MOUNTED;
return -1;
441
442
443
444
445
       // open the host file
       int host_fd = open(host_filename, O_RDONLY);
if (host_fd == -1) {
446
447
       P_ERRNO = P_EOPEN;
448
449
         return -1;
450
451
452
       \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;
453
454
455
         close(host_fd);
456
457
         return -1;
458
459
       // go back to beginning of file for reading
if (lseek(host_fd, 0, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
460
461
462
         close(host_fd);
463
464
         return -1;
465
466
467
       // open the destination file in {\tt PennFAT}
       int pennfat_fd = k_open(pennfat_filename, F_WRITE);
if (pennfat_fd < 0) {
468
469
470
       close(host_fd);
471
         return -1;
472
473
474
       // copy the data into this buffer
475
      uint8_t* buffer = (uint8_t*)malloc(block_size);
       if (!buffer) {
```

```
P_ERRNO = P_EMALLOC;
478
        k_close (pennfat_fd);
479
        close(host_fd);
480
        return -1;
481
482
483
      uint32_t bytes_remaining = host_file_size_in_bytes;
484
      ssize_t bytes_read;
485
486
      // read from host file
      while (bytes_remaining > 0) {
487
       // ensure bytes to read never exceeds the block size
488
489
        ssize_t bytes_to_read =
490
            bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
491
        bytes_read = read(host_fd, buffer, bytes_to_read);
492
        if (bytes_read <= 0) {</pre>
493
494
          break;
495
496
        // write to pennfat_fd using k_write
if (k_write(pennfat_fd, (const char*)buffer, bytes_read) != bytes_read) {
497
498
         free (buffer);
499
500
          k_close(pennfat_fd);
501
          close (host_fd);
502
          return -1;
503
504
505
        bytes_remaining -= bytes_read;
506
507
508
      // check for read error
      if (bytes_read < 0) {
   P_ERRNO = P_EREAD;</pre>
509
510
        free (buffer);
511
       k_close (pennfat_fd);
close (host_fd);
512
513
514
        return -1;
515
516
517
      // otherwise, cleanup and return success
518
      free (buffer);
      k_close(pennfat_fd);
519
     close(host_fd);
520
521
      return 0;
522 }
```

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 527 of file fs_helpers.c.

```
528
529
      if (!is_mounted) {
530
       P_ERRNO = P_EFS_NOT_MOUNTED;
       return -1;
531
532
533
534
      // open the PennFAT file
535
      int pennfat_fd = k_open(pennfat_filename, F_READ);
      if (pennfat_fd < 0) {</pre>
536
     return -1;
537
538
539
540
      // get the pennfat file size
541
      off_t pennfat_file_size_in_bytes = k_lseek(pennfat_fd, 0, SEEK_END);
542
      if (pennfat_file_size_in_bytes == -1) {
543
        k_close(pennfat_fd);
544
        return -1;
545
546
547
      // go back to beginning of file for reading
548
      if (k_lseek(pennfat_fd, 0, SEEK_SET) == -1) {
549
       k_close(pennfat_fd);
550
       return -1;
551
552
553
      // open the host file
554
      int host_fd = open(host_filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
555
      if (host_fd == -1) {
        P_ERRNO = P_EOPEN;
556
        k_close(pennfat_fd);
557
558
        return -1:
559
560
561
      \ensuremath{//} allocate buffer for data transfer
562
      char* buffer = (char*)malloc(block_size);
      if (!buffer) {
  P_ERRNO = P_EMALLOC;
563
564
565
        k_close (pennfat_fd);
566
        close(host_fd);
567
        return -1;
568
569
570
      uint32_t bytes_remaining = pennfat_file_size_in_bytes;
571
      ssize_t bytes_read;
572
573
      // read from PennFAT file and write to host file
574
      while (bytes_remaining > 0) {
575
       // ensure bytes to read never exceeds the block size
576
        ssize_t bytes_to_read =
            bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
577
578
        bytes_read = k_read(pennfat_fd, buffer, bytes_to_read);
579
580
       break;
        if (bytes_read <= 0) {</pre>
581
582
583
584
        if (write(host_fd, buffer, bytes_read) != bytes_read) {
         P_ERRNO = P_EINVAL;
585
586
          free(buffer);
587
          close(host_fd);
588
          k_close(pennfat_fd);
589
          return -1;
590
591
592
        bytes_remaining -= bytes_read;
593
594
595
      // check for read error
      if (bytes_read < 0) {
   P_ERRNO = P_EREAD;</pre>
596
597
598
        free(buffer);
599
        close(host_fd);
600
        k_close(pennfat_fd);
601
        return -1;
602
603
604
      // otherwise, cleanup and return success
605
      free (buffer);
606
      close(host_fd);
607
      k_close(pennfat_fd);
608
     return 0;
609 }
```

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 614 of file fs_helpers.c.

```
615
616
      if (!is mounted) {
617
       P_ERRNO = P_EFS_NOT_MOUNTED;
618
        return -1;
619
620
621
      // open the source file
622
      int source_fd = k_open(source_filename, F_READ);
      if (source_fd < 0) {</pre>
623
624
       return -1;
625
626
      \ensuremath{//} get the source file size
62.7
      off_t source_file_size_in_bytes = k_lseek(source_fd, 0, SEEK_END);
if (source_file_size_in_bytes == -1) {
628
629
630
        k_close(source_fd);
631
        return -1;
632
633
      // move to the beginning of the source file for reading
634
635
      if (k_lseek(source_fd, 0, SEEK_SET) < 0) {</pre>
      k_close(source_fd);
636
637
        return -1;
638
639
      // open the destination file
int dest_fd = k_open(dest_filename, F_WRITE);
if (dest_fd < 0) {</pre>
640
641
642
643
      k_close(source_fd);
644
        return -1;
645
646
      // read from source to destination
647
648
      char* buffer = (char*)malloc(block_size);
649
      if (!buffer) {
650
        P_ERRNO = P_EMALLOC;
651
        k_close(source_fd);
652
        k_close(dest_fd);
653
        return -1;
654
655
656
      uint32_t bytes_remaining = source_file_size_in_bytes;
657
      ssize_t bytes_read;
658
659
      while (bytes_remaining > 0) {
       // make sure the bytes to read doesn't exceed block size
660
661
        ssize_t bytes_to_read =
662
            bytes_remaining < block_size ? bytes_remaining : block_size;</pre>
663
        bytes_read = k_read(source_fd, buffer, bytes_to_read);
664
        if (bytes_read <= 0) {</pre>
665
666
          break:
667
        }
```

```
669
       if (k_write(dest_fd, buffer, bytes_read) != bytes_read) {
670
         free(buffer);
671
          k_close(source_fd);
672
         k_close(dest_fd);
673
          return -1;
674
       }
675
     }
676
677
      // check for read error
     if (bytes_read < 0) {
  free(buffer);</pre>
678
679
        k_close(source_fd);
680
       k_close(dest_fd);
681
682
       return -1;
683
684
     // otherwise, cleanup and return success
685
686
     free (buffer);
687
      k_close(source_fd);
     k_close(dest_fd);
689
     return 0;
690 }
```

4.4.1.7 decrement_fd_ref_count()

Decrements the reference count of a file descriptor.

Parameters

```
fd file descriptor to decrement
```

Returns

new reference count, or -1 on error

If reference count reaches 0, flush field values.

Definition at line 107 of file fs_helpers.c.

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

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 191 of file fs_helpers.c.

```
191
192
      if (!is mounted) {
193
       P_ERRNO = P_EFS_NOT_MOUNTED;
194
        return -1;
195
196
197
      // Start with root directory block (block 1)
      uint16 t current block = 1;
198
      int offset_in_block = 0;
199
     int absolute_offset = 0;
201
      dir_entry_t dir_entry;
202
203
      while (1) {
204
       // Position at the start of current block
205
        if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
206
             -1) {
207
         P_ERRNO = P_ELSEEK;
208
          return -1;
209
210
211
        // reset offset for new block
212
        offset in block = 0;
213
214
        // calculate the absolute offset
215
        absolute_offset = fat_size + (current_block - 1) * block_size;
216
217
        // search current block
218
        while (offset_in_block < block_size) {</pre>
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
          if (dir_entry.name[0] == 0) {
225
226
           break;
227
228
          // check if this is a deleted entry
if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
229
230
           offset_in_block += sizeof(dir_entry);
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
238
           if (entry) {
239
              memcpy(entry, &dir_entry, sizeof(dir_entry));
240
            return absolute_offset; // return the absolute file offset
2.41
242
243
          offset_in_block += sizeof(dir_entry);
```

```
absolute_offset += sizeof(dir_entry);
247
        // if we've reached the end of the current block, check if there's a next
248
249
        // block
        if (fat[current_block] != FAT_EOF) {
250
        current_block = fat[current_block];
251
252
253
254
255
       // no more blocks to search
256
       break;
257
258
259
      // file not found
260 P_ERRNO = P_ENOENT;
261 return -1;
262 }
```

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.
```

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

4.4.1.10 has_executable_permission()

```
int has_executable_permission ( \label{eq:permission} \mbox{int } fd \mbox{ )}
```

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
140
      // validate fd argument
      if (fd < 0 || fd >= MAX_FDS) {
   P_ERRNO = P_EINVAL;
141
142
      return -1;
143
144
145
      // determine whether the file exists
146
      dir_entry_t entry;
int entry_offset = find_file(fd_table[fd].filename, &entry);
147
148
149
      if (entry_offset < 0) {</pre>
      return -1;
150
151
152
      // if it exists, get its permission
if (entry.perm & PERM_EXEC) {
153
154
     return 1;
155
156
157
158
      return 0:
159 }
```

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.

```
if (fd < 0 || fd >= MAX_FDS) {
  P_ERRNO = P_EBADF;
90
91
       return -1;
92
93
    if (!fd_table[fd].in_use) {
95
     P_ERRNO = P_EBADF;
       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.

```
// STDIN (fd 0)
       fd_table[0].in_use = 1;
fd_table[0].ref_count = 1;
strncpy(fd_table[0].filename, "<stdin>", 31);
45
46
47
       fd_table[0].mode = F_READ;
50
       // STDOUT (fd 1)
51
       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
        // STDERR (fd 2)
57
       fd_table[2].in_use = 1;
       strncpy(fd_table[2].filename, "<stderr>", 31);
fd_table[2].mode = F_WRITE; // write-only
fd_table[2].ref_count = 1;
58
59
60
       // other file descriptors (fd 3 and above)
      for (int i = 3; i < MAX_FDS; i++) {
  fd_table[i].in_use = 0;
  fd_table[i].ref_count = 0;
  memset (fd_table[i].filename, 0, sizeof(fd_table[i].filename));
  fd_table[i].size_0.</pre>
64
65
66
           fd_table[i].size = 0;
           fd_table[i].first_block = 0;
69
          fd_table[i].position = 0;
70
           fd_table[i].mode = 0;
71
```

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 399 of file fs helpers.c. 399 if (!is_mounted || entry == NULL || absolute_offset < 0) {
 P_ERRNO = P_EINVAL;</pre> 400 401 402 return -1; 403 404 // free the blocks 405 uint16_t current_block = entry->firstBlock; 406 while (current_block != FAT_FREE && current_block != FAT_EOF) {
 uint16_t next_block = fat[current_block];
 fat[current_block] = FAT_FREE; 407 408 409 410 current_block = next_block; 411 412 413 // mark the entry as deleted in the root directory 414 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; 415 416 417 418 419 420 if (write(fs_fd, &deleted_entry, sizeof(deleted_entry)) != 421 sizeof(deleted_entry)) { P_ERRNO = P_EINVAL; 422 ___KNO = 1 return -1; } 423 424 425 426 // mark the passed entry as deleted 427 entry->name[0] = 1; 428 return 0;

4.4.2 Variable Documentation

4.4.2.1 block_size

429 }

```
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

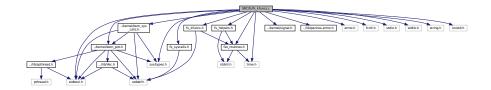
```
int num_fat_blocks [extern]
```

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 526 of file fs kfuncs.c.

```
526
527
       // validate the file descriptor
      if (fd < 0 || fd >= MAX_FDS) {
  P_ERRNO = P_EBADF;
528
529
530
        return -1;
531
532
      // ensure any pending changes are written to disk
533
534
      // update the directory entry with the current file size
      int file_offset = find_file(fd_table[fd].filename, &entry);
535
536
537
      if (file_offset >= 0) {
538
        // update file size if it changed
if (entry.size != fd_table[fd].size) {
539
540
541
           entry.size = fd_table[fd].size;
542
           entry.mtime = time(NULL);
543
          if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
544
545
            return -1;
546
547
548
          if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
549
           P_ERRNO = P_EWRITE;
550
             return -1;
551
552
553
      }
554
555
      // decrement the reference count
556
      decrement_fd_ref_count(fd);
557
558
      return 0;
559 }
```

4.5.1.2 k_ls()

Kernel-level call to list files.

Lists files or file information.

```
Definition at line 666 of file fs_kfuncs.c.
```

```
666
667 if (!is_mounted) {
668    P_ERRNO = P_EFS_NOT_MOUNTED;
669    return -1;
670  }
671
672    // start with root directory block
673    uint16_t current_block = 1;
```

```
dir_entry_t dir_entry;
675
      uint32_t offset = 0;
676
      // if filename is null, list all files in the current directory
677
678
      if (filename == NULL) {
679
        while (1) {
680
          // adjust pointer to beginning of current block
681
           if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
682
            P_ERRNO = P_ELSEEK;
683
684
            return -1;
685
686
687
           offset = 0;
688
689
           // search current block
690
           while (offset < block_size) {</pre>
            if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
    P_ERRNO = P_EREAD;
691
692
693
               return -1;
694
695
             // check if we've reached the end of directory
696
697
             if (dir_entry.name[0] == 0) {
698
               break;
699
700
701
             // skip deleted entries
             if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
702
703
              offset += sizeof(dir_entry);
704
               continue:
705
706
707
             // format permission string
708
             char perm_str[4] = "---";
            if (dir_entry.perm & PERM_READ)
  perm_str[0] = 'r';
709
710
            if (dir_entry.perm & PERM_WRITE)
  perm_str[1] = 'w';
711
712
713
            if (dir_entry.perm & PERM_EXEC)
714
              perm_str[2] = 'x';
715
            // format time
716
717
             struct tm* tm_info = localtime(&dir_entry.mtime);
             char time_str[50];
718
719
             strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);
720
             // print entry details
721
             char buffer[128];
722
723
             int len;
724
             if (dir_entry.firstBlock == 0) {
725
               len = snprintf(buffer, sizeof(buffer), " -%s- %6d %s %s\n",
                                perm_str, dir_entry.size, time_str, dir_entry.name);
726
727
             } else {
               len = snprintf(buffer, sizeof(buffer), "%2d -%s- %6d %s %s\n",
728
                                dir_entry.firstBlock, perm_str, dir_entry.size, time_str, dir_entry.name);
729
730
731
732
             if (len < 0 || len >= (int)sizeof(buffer)) {
   P_ERRNO = P_EUNKNOWN;
733
734
735
               return -1;
736
737
738
             if (k_write(STDOUT_FILENO, buffer, len) != len) {
739
              P_ERRNO = P_EWRITE;
               return -1;
740
741
742
743
             offset += sizeof(dir_entry);
744
745
           // move to the next block if there is one
if (fat[current_block] != FAT_EOF) {
746
747
748
            current block = fat[current block];
749
             continue;
750
751
           \ensuremath{//} no more blocks to search
752
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;
return -1;
759
760
```

```
761
        }
762
763
        if (dir_entry.name[0] == 0) {
        P_ERRNO = P_ENOENT;
764
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
        char perm_str[4] = "---";
if (dir_entry.perm & PERM_READ)
   perm_str[0] = 'r';
if (dir_entry.perm & PERM_WRITE)
   perm_str[1] = 'w';
775
776
777
778
779
780
        if (dir_entry.perm & PERM_EXEC)
781
          perm_str[2] = 'x';
782
783
        // format time
        struct tm* tm_info = localtime(&dir_entry.mtime);
784
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:
791
        if (dir_entry.firstBlock == 0) {
792
          len = snprintf(buffer, sizeof(buffer), " -%s- %6d %s %s\n", perm_str,
793
                           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,
797
                           dir_entry.name);
798
799
800
         if (len < 0 || len >= (int)sizeof(buffer)) {
801
          P_ERRNO = P_EUNKNOWN;
          return -1;
802
803
804
        if (k_write(STDOUT_FILENO, buffer, len) != len) {
806
          P_ERRNO = P_EWRITE;
807
           return -1;
808
      }
809
810
811
      return 0;
812 }
```

4.5.1.3 k_lseek()

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

Repositions the file offset of an open file.

Definition at line 620 of file fs kfuncs.c.

```
620
      // standard file descriptors don't support lseek
621
      if (fd == STDIN_FILENO || fd == STDOUT_FILENO || fd == STDERR_FILENO) {
622
623
      P_ERRNO = P_EINVAL;
624
       return -1;
625
62.6
627
     // validate the file descriptor
628
     if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
       P_ERRNO = P_EBADF;
```

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

4.5.1.4 k open()

Kernel-level call to open a file.

Opens a file with the specified mode.

Definition at line 31 of file fs kfuncs.c.

```
32
     // validate arguments
     if (fname == NULL || *fname == '\0') {
  P_ERRNO = P_EINVAL;
33
34
35
      return -1;
36
37
    if ((mode & (F_READ | F_WRITE | F_APPEND)) == 0) {
38
      P_ERRNO = P_EINVAL;
39
      return -1;
40
41
    // check if the file system is mounted
42
    if (!is_mounted) {
43
44
     P_ERRNO = P_EFS_NOT_MOUNTED;
4.5
       return -1;
    }
46
47
    // get a free file descriptor
48
    int fd = get_free_fd(fd_table);
49
     if (fd < 0) {
     P_ERRNO = P_EFULL; // no free file descriptors
       return -1;
52
    }
53
54
     // check if the file exists
    dir_entry_t entry;
     int file_offset = find_file(fname, &entry);
58
    // file exists
59
60
    if (file_offset >= 0) {
     // check if the file is already open in write mode by another descriptor
61
       if ((mode & (F_WRITE | F_APPEND)) != 0) {
```

```
63
          for (int i = 0; i < MAX_FDS; i++) {</pre>
            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
65
66
67
              return -1;
68
69
            }
70
          }
71
        }
72
73
        // fill in the file descriptor entry
        fd_table[fd].in_use = 1;
74
        fd_table[fd].ref_count++;
75
76
        strncpy(fd_table[fd].filename, fname, 31);
77
        fd_table[fd].filename[31] = ' \setminus 0';
        fd_table[fd].size = entry.size;
fd_table[fd].first_block = entry.firstBlock;
fd_table[fd].mode = mode;
78
79
80
81
        // set the initial position
83
        if (mode & F_APPEND) {
84
          fd_table[fd].position = entry.size;
8.5
        } else {
          fd_table[fd].position = 0;
86
89
        // if mode includes {\tt F\_WRITE} and not {\tt F\_APPEND} , truncate the file
90
        if ((mode & F_WRITE) && !(mode & F_APPEND)) {
91
          // free all blocks except the first one
92
          uint16_t block = entry.firstBlock;
          uint16 t next block;
93
94
95
          if (block != 0 && block != FAT_EOF) {
            next_block = fat[block];
fat[block] = FAT_EOF; // terminate the chain at the first block
96
97
98
            block = next_block;
99
100
              // free the rest of the chain
101
             while (block != 0 && block != FAT_EOF) {
               next_block = fat[block];
fat[block] = FAT_FREE;
102
103
104
               block = next_block;
105
             }
106
           }
107
108
           // update file size to 0
109
           fd_table[fd].size = 0;
110
           entry.size = 0;
           entry.mtime = time(NULL);
111
112
           // update the file system with the truncated file
113
114
           if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
115
             P_ERRNO = P_ELSEEK;
116
             return -1;
117
118
           if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
119
             P_ERRNO = P_EWRITE;
             return -1;
120
121
122
123
      } else {
124
         // file doesn't exist
125
126
         // we can only create it if we are reading the file if (!(mode & F_{wRITE})) {
127
128
         P_ERRNO = P_ENOENT;
129
           return -1;
130
131
132
         // allocate the first block
133
         uint16_t first_block = allocate_block();
134
         if (first_block == 0) {
          P_ERRNO = P_EFULL;
return -1;
135
136
137
138
139
         // create a new file entry
140
         if (add_file_entry(fname, 0, first_block, TYPE_REGULAR, PERM_READ_WRITE) ==
141
              -1) {
           // error code already set by add_file_entry
142
           fat[first_block] = FAT_FREE;
143
144
           return -1;
145
146
147
         // fill in the file descriptor entry
148
         fd_table[fd].in_use = 1;
149
         fd_table[fd].ref_count++;
```

```
150
        strncpy(fd_table[fd].filename, fname, 31);
151
        fd_table[fd].filename[31] = ' \setminus 0';
152
        fd_table[fd].size = 0;
        fd_table[fd].first_block = first_block;
153
154
        fd_table[fd].position = 0;
155
        fd table[fd].mode = mode;
156
157
158 return fd;
159 }
```

4.5.1.5 k read()

Kernel-level call to read a file.

Reads data from an open file.

Definition at line 164 of file fs kfuncs.c.

```
164
       // handle terminal control (if doesn't control, send a STOP signal)
166
       if (fd == STDIN_FILENO && current_running_pcb != NULL) {
167
        if (current_running_pcb->pid != current_fg_pid) {
          s_kill(current_running_pcb->pid, P_SIGSTOP);
168
169
170
      }
171
172
      // handle standard input
173
      return read(STDIN_FILENO, buf, n);
}
      if (fd == STDIN_FILENO) {
174
175
176
177
      // validate inputs
      if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
   P_ERRNO = P_EBADF;
178
179
180
        return -1;
181
      if (buf == NULL || n < 0) {
182
       P_ERRNO = P_EINVAL;
183
184
        return -1;
185
      if (n == 0) {
186
187
        return 0;
188
189
190
      // check if we're at EOF already
191
      if (fd_table[fd].position >= fd_table[fd].size) {
192
       return 0;
193
194
      // determine how many bytes we can actually read
195
196
      uint32_t bytes_to_read = n;
197
      if (fd_table[fd].position + bytes_to_read > fd_table[fd].size) {
198
        bytes_to_read = fd_table[fd].size - fd_table[fd].position;
199
200
201
      \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;
202
203
204
      uint32_t block_offset = fd_table[fd].position % block_size;
205
206
       // 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) {
207
208
209
          // unexpected end of chain
210
          P_ERRNO = P_EINVAL;
211
          return -1;
212
213
        current block = fat[current block];
214
```

```
216
      // now we're at the right block, start reading
217
      uint32_t bytes_read = 0;
218
219
      while (bytes_read < bytes_to_read) {</pre>
220
       // how much data can we read from the current block
221
        uint32_t bytes_left_in_block = block_size - block_offset;
222
        uint32_t bytes_to_read_now =
223
            (bytes_to_read - bytes_read) < bytes_left_in_block
224
               ? (bytes_to_read - bytes_read)
225
                : bytes_left_in_block;
226
       227
228
229
230
         P_ERRNO = P_ELSEEK;
231
         if (bytes_read > 0) {
            fd_table[fd].position += bytes_read;
232
233
           return bytes_read;
234
235
         return -1;
236
237
        \ensuremath{//} read the data from the file
238
        ssize_t read_result = read(fs_fd, buf + bytes_read, bytes_to_read_now);
239
240
        if (read_result <= 0) {</pre>
        P_ERRNO = P_EREAD;
241
         // if we already read some data, return that count
if (bytes_read > 0) {
242
243
           fd_table[fd].position += bytes_read;
2.44
245
           return bytes_read;
246
247
         return -1;
248
249
250
        bytes_read += read_result;
251
        block_offset += read_result;
252
253
        // if we've read all data from this block and still have more to read, go to
254
        // the next block
255
        if (block_offset == block_size && bytes_read < bytes_to_read) {</pre>
256
          if (current_block == FAT_EOF) {
           // unexpected end of chain
257
2.58
           break:
259
260
          current_block = fat[current_block];
261
         block_offset = 0;
262
263
        \ensuremath{//} if we read less than expected, we might have hit EOF
264
       if (read_result < bytes_to_read_now) {</pre>
265
266
         break;
267
268
269
270
      // update file position
     fd_table[fd].position += bytes_read;
271
273
     return bytes_read;
274 }
```

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 564 of file fs_kfuncs.c.
```

```
564
565
if (fname == NULL || *fname == '\0') {
566
    P_ERRNO = P_EINVAL;
567
    return -1;
568
}
```

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

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 279 of file fs kfuncs.c.

```
279
280
      // handle standard output and error
      if (fd == STDOUT_FILENO) {
281
282
       return write(STDOUT_FILENO, str, n);
283
     return write(STDERR_FILENO, str, n);
}
284
285
286
287
      // validate inputs
288
289
      if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
      P_ERRNO = P_EBADF;
290
291
       return -1;
292
293
     if (str == NULL || n < 0) {
       P_ERRNO = P_EINVAL;
```

```
return -1;
296
297
       if (n == 0) {
298
        return 0;
299
300
301
       // check if filesystem is mounted and FAT is valid
302
       if (!is_mounted || fat == NULL) {
303
       P_ERRNO = P_EFS_NOT_MOUNTED;
304
         return -1;
305
306
307
       // get file information
308
       uint16_t current_block = fd_table[fd].first_block;
309
       uint32_t current_position = fd_table[fd].position;
310
       // create a local buffer for block data
311
       char* block_buffer = (char*)malloc(block_size);
if (block_buffer == NULL) {
312
313
314
         P_ERRNO = P_EMALLOC;
315
         return -1;
316
317
       // calculate initial block position
uint32_t block_index = current_position / block_size;
318
319
       uint32_t block_offset = current_position % block_size;
320
321
322
        // if the file doesn't have a first block yet, allocate one
323
       if (current_block == 0) {
         current_block = allocate_block();
324
         if (current_block == 0) {
  P_ERRNO = P_EFULL;
  free(block_buffer);
325
326
327
328
            return -1;
329
         fd_table[fd].first_block = current_block;
330
331
332
333
       // navigate to the appropriate block
       intl6_t prev_block = 0;
if (uint32_t i = 0; i < block_index; i++) {
   if (current_block == 0 || current_block == FAT_EOF ||
      current_block >= fat_size / 2) {
      // reached the end of chain prematurely, need to allocate a new block
334
335
336
337
338
            uint16_t new_block = allocate_block();
339
340
            if (new_block == 0) {
341
             P ERRNO = P EFULL:
              free(block_buffer);
342
              return -1;
343
344
345
346
            // update the chain
347
            if (prev_block != 0 && prev_block < fat_size / 2) {</pre>
348
              fat[prev_block] = new_block;
349
            } else {
350
              // if there's no previous block, this must be the first one
351
              fd_table[fd].first_block = new_block;
352
353
354
           current_block = new_block;
355
356
357
         prev_block = current_block;
358
359
          // validate the block number before accessing FAT
360
         if (current_block >= fat_size / 2) {
           P_ERRNO = P_EINVAL:
361
            free(block_buffer);
362
363
            return -1:
364
365
366
         current_block = fat[current_block];
367
368
       // if we ended up without a valid block, go back to the last valid one if (current_block == 0 || current_block == FAT_EOF ||
369
370
371
            current_block >= fat_size / 2) {
372
          if (prev_block != 0 && prev_block < fat_size / 2) {</pre>
373
            uint16_t new_block = allocate_block();
            if (new_block == 0) {
P_ERRNO = P_EFULL;
374
375
              free(block_buffer);
376
              return -1;
377
378
379
            fat[prev_block] = new_block;
380
381
            current block = new block;
```

```
382
        } else {
383
         P_ERRNO = P_EINVAL;
384
          free(block_buffer);
385
          return -1;
386
387
      }
388
389
      // start writing data
390
      uint32_t bytes_written = 0;
391
392
      while (bytes_written < n) {</pre>
393
        // validate current block
        if (current_block == 0 || current_block == FAT_EOF ||
394
            current_block >= fat_size / 2) {
395
396
          P_ERRNO = P_EINVAL;
397
          break;
398
399
400
        // how much can we write to this block
        uint32_t space_in_block = block_size - block_offset;
401
402
        uint32_t bytes_to_write = (n - bytes_written) < space_in_block</pre>
403
                                        ? (n - bytes_written)
404
                                        : space_in_block;
405
406
        // position in filesystem
407
        off_t block_position = fat_size + (current_block - 1) * block_size;
408
409
         // if we're not writing a full block or not starting at the beginning, we
410
        // need to read-modify-write
        if (bytes_to_write < block_size || block_offset > 0) {
411
412
          // read the current block
413
          if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
414
           P_ERRNO = P_ELSEEK;
415
            break;
416
417
          // read the current block data
418
419
          ssize_t read_result = read(fs_fd, block_buffer, block_size);
420
          if (read_result < 0) {</pre>
421
           P_ERRNO = P_EREAD;
422
            break;
          }
423
424
425
          // copy the new data into the block buffer
          memcpy(block_buffer + block_offset, str + bytes_written, bytes_to_write);
426
427
428
          // seek back to write the modified block
          if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
429
430
431
            break:
432
433
434
          // write the full block back
435
          ssize_t write_result = write(fs_fd, block_buffer, block_size);
          if (write_result != block_size) {
436
            P_ERRNO = P_EWRITE;
437
438
             // we might have a partial write, but that's hard to handle correctly
439
            break:
440
        } else {
441
          // we're writing a full block from the beginning
if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
442
443
444
            P_ERRNO = P_ELSEEK;
445
           break;
446
447
          ssize_t write_result = write(fs_fd, str + bytes_written, bytes_to_write);
448
          if (write_result != bytes_to_write) {
449
            P_ERRNO = P_EWRITE;
450
451
            break;
452
453
454
        // update counters
455
456
        bytes written += bytes to write;
457
        block_offset = (block_offset + bytes_to_write) % block_size;
458
459
        // if we've filled this block and still have more to write, go to the next
460
        // block
461
        if (block offset == 0 && bytes written < n) {
          // validate current block before accessing FAT
462
          if (current_block >= fat_size / 2) {
463
464
            P_ERRNO = P_EINVAL;
465
            break;
466
          }
467
468
          // check if there's a next block
```

```
if (fat[current_block] == FAT_EOF) {
469
470
            // allocate a new block
471
             uint16_t new_block = allocate_block();
            if (new_block == 0) {
   P_ERRNO = P_EFULL;
472
473
474
               break:
475
476
            // Update the FAT safely
if (current_block < fat_size / 2) {
  fat[current_block] = new_block;</pre>
477
478
479
480
             } else {
               P_ERRNO = P_EINVAL;
481
              break;
482
483
484
485
             current_block = new_block;
486
         } else {
487
             current_block = fat[current_block];
488
489
490
491
       // free the block buffer
492
493
      free (block_buffer);
494
495
      // update file position
496
      fd_table[fd].position += bytes_written;
497
498
      // update file size if needed
      if (fd_table[fd].position > fd_table[fd].size) {
499
500
        fd_table[fd].size = fd_table[fd].position;
501
502
         // update the directory entry
        int dir_entry_t entry;
int dir_offset = find_file(fd_table[fd].filename, &entry);
if (dir_offset >= 0) {
  entry.size = fd_table[fd].size;
503
504
505
506
507
          entry.mtime = time(NULL);
508
509
          if (lseek(fs_fd, dir_offset, SEEK_SET) == -1) {
           P_ERRNO = P_ELSEEK;
return -1;
510
511
512
513
          if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
514
             P_ERRNO = P_EWRITE;
515
             return -1;
516
        }
517
518 }
519
520
      return bytes_written;
521 }
```

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.

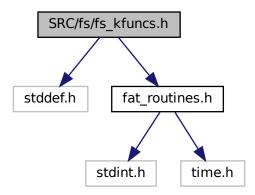
4.5.2.2 current_running_pcb

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

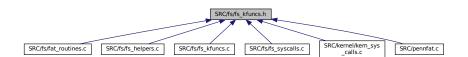
Definition at line 38 of file scheduler.c.

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 526 of file fs_kfuncs.c.

```
// validate the file descriptor
if (fd < 0 || fd >= MAX_FDS) {
   P_ERRNO = P_EBADF;
527
528
529
         return -1;
530
531
532
533
       \ensuremath{//} ensure any pending changes are written to disk
534
       // update the directory entry with the current file size
      dir_entry_t entry;
int file_offset = find_file(fd_table[fd].filename, &entry);
535
536
538
       if (file_offset >= 0) {
        // update file size if it changed
if (entry.size != fd_table[fd].size) {
  entry.size = fd_table[fd].size;
539
540
541
            entry.mtime = time(NULL);
542
543
544
            if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
545
            P_ERRNO = P_ELSEEK;
546
              return -1;
547
548
            if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
              P_ERRNO = P_EWRITE;
550
              return -1;
551
552
         }
553
554
555
       // decrement the reference count
      decrement_fd_ref_count(fd);
557
558
      return 0;
559 }
```

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 666 of file fs_kfuncs.c.

```
667
      if (!is mounted) {
        P_ERRNO = P_EFS_NOT_MOUNTED;
668
        return -1;
669
670
671
672
      // start with root directory block
673
     uint16_t current_block = 1;
674
     dir_entry_t dir_entry;
675
     uint32 t offset = 0;
676
677
      // if filename is null, list all files in the current directory
678
      if (filename == NULL) {
679
        while (1) {
680
         // adjust pointer to beginning of current block
          if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
681
682
              -1) {
683
            P_ERRNO = P_ELSEEK;
684
            return -1;
685
686
          offset = 0;
687
688
689
          // search current block
690
          while (offset < block_size) {</pre>
691
            if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
692
              P_ERRNO = P_EREAD;
              return -1:
693
694
695
696
            // check if we've reached the end of directory
697
            if (dir_entry.name[0] == 0) {
698
              break;
699
700
701
            // skip deleted entries
            if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
702
703
              offset += sizeof(dir_entry);
704
              continue;
705
706
707
            // format permission string
            char perm_str[4] = "---
708
709
            if (dir_entry.perm & PERM_READ)
710
              perm_str[0] = 'r';
           if (dir_entry.perm & PERM_WRITE)
perm_str[1] = 'w';
if (dir_entry.perm & PERM_EXEC)
perm_str[2] = 'x';
711
712
713
714
715
716
717
            struct tm* tm_info = localtime(&dir_entry.mtime);
718
            char time_str[50];
            strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);
719
720
721
            // print entry details
722
            char buffer[128];
723
            int len;
724
            if (dir_entry.firstBlock == 0) {
              len = snprintf(buffer, sizeof(buffer), " -%s- %6d %s %s\n",
725
726
                              perm_str, dir_entry.size, time_str, dir_entry.name);
727
            } else {
728
              len = snprintf(buffer, sizeof(buffer), "%2d -%s- %6d %s %s\n",
729
                              dir_entry.firstBlock, perm_str, dir_entry.size,
730
                              time_str, dir_entry.name);
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;
              return -1;
```

```
741
            }
742
743
            offset += sizeof(dir_entry);
          }
744
745
746
           // move to the next block if there is one
747
          if (fat[current_block] != FAT_EOF) {
748
            current_block = fat[current_block];
749
            continue;
750
751
          // no more blocks to search
752
753
          break;
754
      } else {
755
        // find and display specific file
int file_offset = find_file(filename, &dir_entry);
if (file_offset < 0) {</pre>
756
757
758
759
          P_ERRNO = P_ENOENT;
760
          return -1;
761
762
763
        if (dir_entry.name[0] == 0) {
764
          P_ERRNO = P_ENOENT;
765
          return -1;
766
767
768
         // skip deleted entries
        if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
769
770
          P_ERRNO = P_ENOENT;
771
          return -1;
772
773
774
        // format permission string
        if (dir_entry.perm & PERM_WRITE)
print(dir_entry.perm & PERM_WRITE)
775
776
777
778
779
          perm_str[1] = 'w';
        if (dir_entry.perm & PERM_EXEC)
  perm_str[2] = 'x';
780
781
782
        // format time
783
        struct tm* tm_info = localtime(&dir_entry.mtime);
784
785
        char time_str[50];
786
        strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);
787
        // print entry details
788
        char buffer[128];
789
        int len;
790
791
        if (dir_entry.firstBlock == 0) {
792
          len = snprintf(buffer, sizeof(buffer), " -%s- %6d %s %s\n", perm_str,
793
                           dir_entry.size, time_str, dir_entry.name);
794
        } else {
          len = snprintf(buffer, sizeof(buffer), "%2d -%s- %6d %s %s\n",
795
796
                           dir_entry.firstBlock, perm_str, dir_entry.size, time_str,
797
                           dir_entry.name);
798
799
        if (len < 0 || len >= (int)sizeof(buffer)) {
800
        P_ERRNO = P_EUNKNOWN;
801
          return -1;
802
803
804
805
        if (k_write(STDOUT_FILENO, buffer, len) != len) {
806
         P_ERRNO = P_EWRITE;
           return -1;
807
        }
808
      }
809
810
811
      return 0;
812 }
```

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

620

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 620 of file fs_kfuncs.c.

```
621
      // standard file descriptors don't support lseek
622
      if (fd == STDIN_FILENO || fd == STDOUT_FILENO || fd == STDERR_FILENO) {
623
        P_ERRNO = P_EINVAL;
624
        return -1;
625
626
627
      // validate the file descriptor
      if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
629
      P_ERRNO = P_EBADF;
630
        return -1;
631
632
633
      // calculate new position based on whence
634
     int32_t new_position;
635
636
     switch (whence) {
637
       case SEEK_SET:
        new_position = offset;
638
639
         break;
       case SEEK_CUR:
640
641
        new_position = fd_table[fd].position + offset;
642
        case SEEK_END:
643
         new_position = fd_table[fd].size + offset;
644
645
          break;
646
        default:
647
         P_ERRNO = P_EINVAL;
648
          return -1;
649
650
     // check if new position is valid
if (new_position < 0) {</pre>
651
652
653
      P_ERRNO = P_EINVAL;
654
       return -1;
655
656
     // update file position
fd_table[fd].position = new_position;
657
658
659
660 return new_position;
661 }
```

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 31 of file fs kfuncs.c.
```

```
// validate arguments
32
      if (fname == NULL || *fname == '\0') {
33
      P_ERRNO = P_EINVAL;
34
35
        return -1;
36
37
     if ((mode & (F_READ | F_WRITE | F_APPEND)) == 0) {
38
        P_ERRNO = P_EINVAL;
        return -1;
39
     }
40
41
     // check if the file system is mounted
43
     if (!is_mounted) {
       P_ERRNO = P_EFS_NOT_MOUNTED;
44
45
       return -1;
46
     // get a free file descriptor
    int fd = get_free_fd(fd_table);
if (fd < 0) {</pre>
50
51
       P_ERRNO = P_EFULL; // no free file descriptors
       return -1;
52
53
55
     // check if the file exists
     dir_entry_t entry;
56
     int file_offset = find_file(fname, &entry);
57
58
     // file exists
59
     if (file_offset >= 0) {
60
        // check if the file is already open in write mode by another descriptor
       if ((mode & (F_WRITE | F_APPEND))) != 0) {
   for (int i = 0; i < MAX_FDS; i++) {</pre>
63
            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
64
65
69
70
          }
72
        // fill in the file descriptor entry
```

```
74
        fd_table[fd].in_use = 1;
75
        fd_table[fd].ref_count++;
76
        strncpy(fd_table[fd].filename, fname, 31);
77
        fd_table[fd].filename[31] = ' \setminus 0';
78
        fd_table[fd].size = entry.size;
fd_table[fd].first_block = entry.firstBlock;
79
        fd_table[fd].mode = mode;
80
81
82
        // set the initial position
        if (mode & F_APPEND) {
  fd_table[fd].position = entry.size;
83
84
85
        } else {
          fd_table[fd].position = 0;
86
87
88
89
        // if mode includes {\tt F\_WRITE} and not {\tt F\_APPEND} , truncate the file
        if ((mode & F_WRITE) && !(mode & F_APPEND)) {
   // free all blocks except the first one
   uint16_t block = entry.firstBlock;
90
91
92
          uint16_t next_block;
94
          if (block != 0 && block != FAT_EOF) {
95
            next_block = fat[block];
fat[block] = FAT_EOF; // terminate the chain at the first block
96
97
98
             block = next_block;
100
              // free the rest of the chain
101
              while (block != 0 && block != FAT_EOF) {
                next_block = fat[block];
fat[block] = FAT_FREE;
102
103
104
                block = next block;
105
              }
106
107
108
            // update file size to 0
            fd_table[fd].size = 0;
109
110
            entry.size = 0;
            entry.mtime = time(NULL);
111
112
113
            // update the file system with the truncated file
114
            if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
             P_ERRNO = P_ELSEEK;
return -1;
115
116
117
           if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
118
119
              P_ERRNO = P_EWRITE;
120
              return -1;
121
122
123
       } else {
124
         // file doesn't exist
125
126
         \ensuremath{//} we can only create it if we are reading the file
         if (!(mode & F_WRITE)) {
  P_ERRNO = P_ENOENT;
127
128
           return -1;
129
130
131
132
         // allocate the first block
133
         uint16_t first_block = allocate_block();
         if (first_block == 0) {
P_ERRNO = P_EFULL;
134
135
136
           return -1;
137
138
139
         // create a new file entry
         if (add_file_entry(fname, 0, first_block, TYPE_REGULAR, PERM_READ_WRITE) ==
140
141
              -1) {
            // error code already set by add_file_entry
142
           fat[first_block] = FAT_FREE;
143
144
           return -1;
145
146
         // fill in the file descriptor entry
147
         fd_table[fd].in_use = 1;
fd_table[fd].ref_count++;
148
149
150
         strncpy(fd_table[fd].filename, fname, 31);
151
         fd_table[fd].filename[31] = ' \setminus 0';
         fd_table[fd].size = 0;
fd_table[fd].first_block = first_block;
152
153
         fd_table[fd].position = 0;
154
         fd_table[fd].mode = mode;
155
156
157
158
      return fd;
159 }
```

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 164 of file fs_kfuncs.c.

```
164
         // handle terminal control (if doesn't control, send a STOP signal)
165
        if (fd == STDIN_FILENO && current_running_pcb != NULL) {
   if (current_running_pcb->pid != current_fg_pid) {
166
167
168
             s_kill(current_running_pcb->pid, P_SIGSTOP);
169
170
171
        // handle standard input
if (fd == STDIN_FILENO) {
172
173
174
          return read(STDIN_FILENO, buf, n);
175
176
        // validate inputs
if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
177
178
179
         P_ERRNO = P_EBADF;
180
          return -1;
181
        if (buf == NULL || n < 0) {</pre>
182
         P_ERRNO = P_EINVAL;
183
           return -1;
184
185
186
       if (n == 0) {
187
          return 0;
188
189
        // check if we're at EOF already
if (fd_table[fd].position >= fd_table[fd].size) {
190
191
192
          return 0;
193
194
       // 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;
195
196
197
198
199
```

```
200
201
      // find the block containing the current position
      uint16_t current_block = fd_table[fd].first_block;
uint32_t block_index = fd_table[fd].position / block_size;
uint32_t block_offset = fd_table[fd].position % block_size;
202
203
2.04
205
206
      // navigate to the correct block in the chain
207
      for (uint32_t i = 0; i < block_index; i++) {</pre>
208
       if (current_block == 0 || current_block == FAT_EOF) {
209
           \ensuremath{//} unexpected end of chain
          P_ERRNO = P_EINVAL;
210
          return -1;
211
212
213
        current_block = fat[current_block];
214
215
      // now we're at the right block, start reading
216
217
      uint32_t bytes_read = 0;
218
219
      while (bytes_read < bytes_to_read) {</pre>
220
        // how much data can we read from the current block
221
        uint32_t bytes_left_in_block = block_size - block_offset;
2.2.2
        uint32_t bytes_to_read_now =
             (bytes_to_read - bytes_read) < bytes_left_in_block
? (bytes_to_read - bytes_read)</pre>
223
224
225
                 : bytes_left_in_block;
226
227
         // seek to the right position in the file
        228
229
          P_ERRNO = P_ELSEEK;
230
231
          if (bytes_read > 0) {
232
            fd_table[fd].position += bytes_read;
233
             return bytes_read;
234
235
          return -1;
236
        }
237
238
         // read the data from the file
239
        ssize_t read_result = read(fs_fd, buf + bytes_read, bytes_to_read_now);
240
         if (read_result <= 0) {</pre>
          P_ERRNO = P_EREAD;
2.41
          // if we already read some data, return that count if (bytes_read > 0) {
2.42
243
            fd_table[fd].position += bytes_read;
244
245
             return bytes_read;
246
2.47
          return -1;
248
249
250
        bytes_read += read_result;
251
        block_offset += read_result;
252
253
         // if we've read all data from this block and still have more to read, go to
254
        // the next block
        if (block_offset == block_size && bytes_read < bytes_to_read) {</pre>
255
256
          if (current_block == FAT_EOF) {
257
             // unexpected end of chain
258
            break;
259
          current_block = fat[current_block];
260
          block_offset = 0;
261
262
263
264
         // if we read less than expected, we might have hit EOF
265
        if (read_result < bytes_to_read_now) {</pre>
266
          break;
        }
267
268
269
270
      // update file position
      fd_table[fd].position += bytes_read;
271
2.72
273
      return bytes_read;
274 }
```

4.6.2.6 k unlink()

```
int k\_unlink ( const char * fname )
```

Removes a file from the file system.

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

Parameters

fname The name of the file to remove.

Returns

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

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

Removes a file from the file system.

```
Definition at line 564 of file fs_kfuncs.c.
```

```
if (fname == NULL || *fname == '\0') {
565
       P_ERRNO = P_EINVAL;
566
567
        return -1;
568
569
570
      if (!is_mounted) {
571
      P_ERRNO = P_EFS_NOT_MOUNTED;
572
        return -1;
573
574
575
      \ensuremath{//} check if file is currently open by any process
576
      for (int i = 0; i < MAX_FDS; i++) {</pre>
577
       if (fd_table[i].in_use && strcmp(fd_table[i].filename, fname) == 0) {
578
         P_ERRNO = P_EBUSY;
579
          return -1;
580
       }
581
582
583
      // find the file in directory
      dir_entry_t entry;
int file_offset = find_file(fname, &entry);
if (file_offset < 0) {</pre>
584
585
586
587
      P_ERRNO = P_ENOENT;
588
        return -1;
589
590
      \ensuremath{//} mark the directory entry as deleted (set first byte to 1)
591
592
      entry.name[0] = 1;
594
      \ensuremath{//} write the modified directory entry back
595
      if (lseek(fs_fd, file_offset, SEEK_SET) == -1) {
      P_ERRNO = P_ELSEEK;
return -1;
596
597
598
599
      if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
      P_ERRNO = P_EWRITE;
return -1;
600
601
602
603
      // free all blocks in the file chain
604
      uint16_t current_block = entry.firstBlock;
605
      uint16_t next_block;
607
608
      while (current_block != FAT_FREE && current_block != FAT_EOF) {
609
       next_block = fat[current_block];
        fat[current_block] = FAT_FREE;
610
611
        current_block = next_block;
613
614
      return 0;
615 }
```

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 279 of file fs kfuncs.c.

```
280
        // handle standard output and error
281
       if (fd == STDOUT_FILENO) {
282
         return write(STDOUT_FILENO, str, n);
283
284
       if (fd == STDERR_FILENO) {
         return write(STDERR_FILENO, str, n);
285
286
287
288
       // validate inputs
       if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
   P_ERRNO = P_EBADF;
289
290
291
         return -1;
292
293
       if (str == NULL || n < 0) {</pre>
        P_ERRNO = P_EINVAL;
294
295
         return -1;
296
297
       if (n == 0) {
298
         return 0;
299
300
       // check if filesystem is mounted and FAT is valid
if (!is_mounted || fat == NULL) {
   P_ERRNO = P_EFS_NOT_MOUNTED;
301
302
303
304
         return -1;
305
306
       // get file information
307
       uint16_t current_block = fd_table[fd].first_block;
uint32_t current_position = fd_table[fd].position;
308
309
310
311
       // create a local buffer for block data
       char* block_buffer = (char*)malloc(block_size);
if (block_buffer == NULL) {
312
313
314
         P_ERRNO = P_EMALLOC;
         return -1;
315
316
```

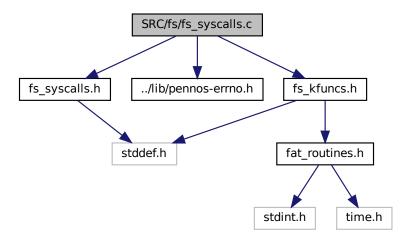
```
318
      // calculate initial block position
      uint32_t block_index = current_position / block_size;
uint32_t block_offset = current_position % block_size;
319
320
321
      // if the file doesn't have a first block yet, allocate one
if (current_block == 0) {
322
323
        current_block = allocate_block();
324
325
         if (current_block == 0) {
326
          P_ERRNO = P_EFULL;
327
           free(block_buffer);
328
           return -1;
329
330
         fd_table[fd].first_block = current_block;
331
332
333
      \ensuremath{//} navigate to the appropriate block
334
      uint16_t prev_block = 0;
      for (uint32_t i = 0; i < block_index; i++) {</pre>
335
        if (current_block == 0 || current_block == FAT_EOF ||
336
             current_block >= fat_size / 2) {
337
338
           // reached the end of chain prematurely, need to allocate a new block
339
           uint16_t new_block = allocate_block();
           if (new_block == 0) {
  P ERRNO = P EFULL;
340
341
342
             free(block_buffer);
343
             return -1;
344
345
           // update the chain
if (prev_block != 0 && prev_block < fat_size / 2) {</pre>
346
347
348
             fat[prev_block] = new_block;
349
           } else {
350
            // if there's no previous block, this must be the first one
351
             fd_table[fd].first_block = new_block;
352
           }
353
354
          current block = new block;
355
356
357
         prev_block = current_block;
358
359
         // validate the block number before accessing FAT
         if (current_block >= fat_size / 2) {
   P_ERRNO = P_EINVAL;
360
361
           free(block_buffer);
362
363
           return -1;
364
365
366
         current_block = fat[current_block];
367
368
369
       // 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>
370
371
372
373
           uint16_t new_block = allocate_block();
374
           <u>if</u> (new_block == 0) {
375
             P_ERRNO = P_EFULL;
376
             free(block_buffer);
377
             return -1;
378
379
380
           fat[prev_block] = new_block;
           current_block = new_block;
381
         } else {
382
383
           P_ERRNO = P_EINVAL;
           free (block_buffer);
384
           return -1;
385
386
387
388
389
      // start writing data
390
      uint32_t bytes_written = 0;
391
392
      while (bytes written < n) {
393
        // validate current block
394
         if (current_block == 0 || current_block == FAT_EOF ||
395
              current_block >= fat_size / 2) {
           P_ERRNO = P_EINVAL;
396
397
           break:
398
399
400
         // how much can we write to this block
401
         uint32_t space_in_block = block_size - block_offset;
         uint32_t bytes_to_write = (n - bytes_written) < space_in_block</pre>
402
403
                                            ? (n - bytes written)
404
                                            : space_in_block;
```

```
405
        // position in filesystem
406
407
        off_t block_position = fat_size + (current_block - 1) * block_size;
408
409
        // if we're not writing a full block or not starting at the beginning, we
        // need to read-modify-write
410
        if (bytes_to_write < block_size || block_offset > 0) {
411
412
          // read the current block
413
          if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
414
            P_ERRNO = P_ELSEEK;
415
            break:
416
417
418
          // read the current block data
419
          ssize_t read_result = read(fs_fd, block_buffer, block_size);
          if (read_result < 0) {</pre>
420
421
            P_ERRNO = P_EREAD;
422
            break;
423
424
425
          // copy the new data into the block buffer
426
          memcpy(block_buffer + block_offset, str + bytes_written, bytes_to_write);
42.7
          // seek back to write the modified block
if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
428
429
           P_ERRNO = P_ELSEEK;
430
431
432
433
434
          // write the full block back
435
          ssize t write result = write(fs fd, block buffer, block size);
436
          if (write_result != block_size) {
437
           P_ERRNO = P_EWRITE;
438
            // we might have a partial write, but that's hard to handle correctly
439
            break;
440
        } else {
441
          // we're writing a full block from the beginning
443
          if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
444
           P_ERRNO = P_ELSEEK;
445
            break;
          }
446
447
448
          ssize_t write_result = write(fs_fd, str + bytes_written, bytes_to_write);
          if (write_result != bytes_to_write) {
450
            P_ERRNO = P_EWRITE;
451
            break;
452
          }
453
454
455
        // update counters
456
        bytes_written += bytes_to_write;
457
        block_offset = (block_offset + bytes_to_write) % block_size;
458
        // if we've filled this block and still have more to write, go to the next
459
460
        // block
        if (block_offset == 0 && bytes_written < n) {</pre>
461
462
          // validate current block before accessing FAT
463
          if (current_block >= fat_size / 2) {
464
            P_ERRNO = P_EINVAL;
465
            break;
466
467
468
          // check if there's a next block
469
          if (fat[current_block] == FAT_EOF) {
470
            // allocate a new block
471
            uint16_t new_block = allocate_block();
if (new_block == 0) {
472
             P_ERRNO = P_EFULL;
473
474
              break;
475
476
            // Update the FAT safely
if (current_block < fat_size / 2) {</pre>
477
478
479
              fat[current_block] = new_block;
480
481
              P_ERRNO = P_EINVAL;
482
              break;
483
484
485
            current_block = new_block;
486
          } else {
487
            current_block = fat[current_block];
488
489
       }
490
      }
491
```

```
492
       // free the block buffer
493
       free(block_buffer);
494
495
       \label{eq:continuous} \mbox{// update file position}
496
       fd_table[fd].position += bytes_written;
497
498
       // update file size if needed
499
       if (fd_table[fd].position > fd_table[fd].size) {
500
        fd_table[fd].size = fd_table[fd].position;
501
502
         // update the directory entry
         dir_entry_t entry;
int dir_offset = find_file(fd_table[fd].filename, &entry);
if (dir_offset >= 0) {
503
504
505
506
           entry.size = fd_table[fd].size;
           entry.mtime = time(NULL);
507
508
           if (lseek(fs_fd, dir_offset, SEEK_SET) == -1) {
   P_ERRNO = P_ELSEEK;
509
510
511
             return -1;
512
           if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
513
             P_ERRNO = P_EWRITE;
return -1;
514
515
516
517
518
519
520
      return bytes_written;
521 }
```

4.7 SRC/fs/fs syscalls.c File Reference

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

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

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.

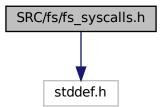
Definition at line 33 of file fs_syscalls.c.

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

4.8 SRC/fs/fs_syscalls.h File Reference

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

Include dependency graph for fs_syscalls.h:



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



Macros

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

Functions

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

Opens a file with the specified access mode.

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

Reads data from an open file.

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

Writes data to an open file.

• int s_close (int fd)

Closes an open file descriptor.

• int s_unlink (const char *fname)

Removes a file from the file system.

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

Repositions the file offset of an open file.

• int s_ls (const char *filename)

Lists files in the current directory or displays file information.

4.8.1 Macro Definition Documentation

4.8.1.1 STDERR_FILENO

```
#define STDERR_FILENO 2
```

Definition at line 18 of file fs_syscalls.h.

4.8.1.2 STDIN FILENO

```
#define STDIN_FILENO 0
```

Definition at line 16 of file fs_syscalls.h.

4.8.1.3 STDOUT_FILENO

```
#define STDOUT_FILENO 1
```

Definition at line 17 of file fs_syscalls.h.

4.8.2 Function Documentation

4.8.2.1 s_close()

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

Closes an open file descriptor.

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

Parameters

fd The file descriptor to close.

Returns

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

• P_EBADF: fd is not a valid file descriptor.

Closes an open file descriptor.

This is a wrapper around the kernel function k_close.

Definition at line 42 of file fs_syscalls.c.

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

4.8.2.2 s_ls()

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

Lists files in the current directory or displays file information.

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

Parameters

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

Returns

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

• P ENOENT: The specified file does not exist.

Lists files in the current directory or displays file information.

This is a wrapper around the kernel function k ls.

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

```
69 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 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()

```
int s_unlink ( \label{eq:const_char} \mbox{const_char} \ * \ \mbox{\it fname} \ )
```

Removes a file from the file system.

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

Parameters

fname	The name of the file to remove.
-------	---------------------------------

Returns

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

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

Removes a file from the file system.

This is a wrapper around the kernel function k_unlink.

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

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

4.8.2.7 s_write()

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

Writes data to an open file.

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

Parameters

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

Returns

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

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

Writes data to an open file.

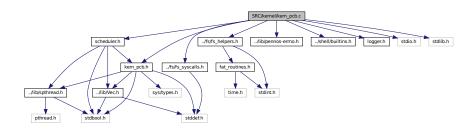
This is a wrapper around the kernel function k_write.

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

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 "../shell/builtins.h"
#include "logger.h"
#include "scheduler.h"
#include "stdio.h"
#include "stdlib.h"
```

Include dependency graph for kern_pcb.c:



Functions

void free pcb (void *pcb)

Free resources associated with a PCB.

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

Initializes a PCB with the given parameters.

void remove_child_in_parent (pcb_t *parent, pcb_t *child)

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

pcb_t * k_proc_create (pcb_t *parent, int priority)

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

void k_proc_cleanup (pcb_t *proc)

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

Variables

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

4.9.1 Function Documentation

4.9.1.1 create_pcb()

Initializes a PCB with the given parameters.

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

Definition at line 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
62
                                                   // double free
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 152 of file kern_pcb.c.

```
// if proc has parent (i.e. isn't init) then remove it from parent's child
153
154
     pcb_t* par_pcb = get_pcb_in_queue(&current_pcbs, proc->par_pid);
155
     if (par_pcb != NULL) {
156
157
       remove_child_in_parent(par_pcb, proc);
158
159
       P_ERRNO = P_ENULL;
160
       return;
161
162
163
     // if proc has children, remove them and assign them to init parent
     if (vec_len(&proc->child_pcbs) > 0) {
164
       // retrieve the init process
165
166
       pcb_t* init_pcb =
           get_pcb_in_queue(&current_pcbs, 1); // init process has pid 1
167
168
169
       while (vec_len(&proc->child_pcbs) > 0) {
         pcb_t* curr_child = vec_get(&proc->child_pcbs, 0);
171
          vec_push_back(&init_pcb->child_pcbs, curr_child);
172
         vec_erase_no_deletor(&proc->child_pcbs, 0); // don't free in erase
         173
174
175
176
177
178
      // decr reference counts + close files if necessary
179
     for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
  if (proc->fd_table[i] != -1) {
180
181
            (decrement_fd_ref_count(proc->fd_table[i]) == 0) {
182
183
           if (s_close(proc->fd_table[i]) == -1) {
184
             u_perror("closing on a non-valid fd");
185
           }
186
         }
187
       }
188
     }
```

```
190
      // cancel + join this thread
191
      spthread_cancel(proc->thread_handle);
192
      spthread_continue(proc->thread_handle);
193
      spthread_suspend(proc->thread_handle);
194
      spthread_join(proc->thread_handle, NULL);
195
196
      // delete this process from any queue it's in + free it
197
      delete_process_from_all_queues(proc);
198 free_pcb(proc);
199 }
```

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 95 of file kern_pcb.c.

```
if (parent == NULL) { // init creation case
97
       pcb_t* init = create_pcb(1, 0, 0, 0, 1);
       if (init == NULL) {
  P_ERRNO = P_ENULL;
98
99
          return NULL;
100
101
102
        init->fd_table[0] = STDIN_FILENO;
        init->fd_table[1] = STDOUT_FILENO;
init->fd_table[2] = STDERR_FILENO;
103
104
105
        for (int i = 3; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {</pre>
          init->fd_table[i] = -1;
106
107
108
109
        increment_fd_ref_count(STDIN_FILENO);
110
        increment_fd_ref_count(STDOUT_FILENO);
111
        increment_fd_ref_count(STDERR_FILENO);
112
113
        current running pcb = init;
114
        put_pcb_into_correct_queue(init);
115
         vec_push_back(&current_pcbs, init);
116
        return init;
117
118
      pcb_t* child = create_pcb(next_pid++, parent->pid, priority, parent->input_fd,
119
                                  parent->output_fd);
120
121
      if (child == NULL) {
122
       P_ERRNO = P_ENULL;
123
        return NULL;
124
125
126
      // copy parent's fd table
127
      for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {</pre>
128
       child->fd_table[i] = parent->fd_table[i];
129
130
      for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
  if (child->fd_table[i] != -1) {
131
132
133
          increment_fd_ref_count(child->fd_table[i]);
134
135
136
      // update parent as needed
137
      vec_push_back(&parent->child_pcbs, child);
138
139
140
      // add to appropriate queue
141
      put_pcb_into_correct_queue(child);
142
      vec_push_back(&current_pcbs, child);
143
144
      return child;
145 }
```

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.

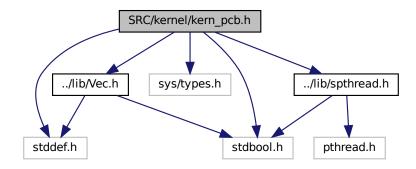
4.9.2.3 next_pid

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

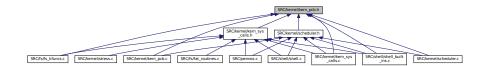
Definition at line 17 of file kern_pcb.c.

4.10 SRC/kernel/kern_pcb.h File Reference

```
#include <stdbool.h>
#include <stddef.h>
#include <sys/types.h>
#include "../lib/Vec.h"
#include "../lib/spthread.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 16 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;
     ret_pcb->output_fd = output_fd;
58
      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
69
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 152 of file kern_pcb.c.

```
152
      // if proc has parent (i.e. isn't init) then remove it from parent's child
154
      // list
      pcb_t* par_pcb = get_pcb_in_queue(&current_pcbs, proc->par_pid);
155
      if (par_pcb != NULL) {
156
157
       remove_child_in_parent(par_pcb, proc);
     } else {
158
      P_ERRNO = P_ENULL;
159
160
        return;
161
162
163
      // if proc has children, remove them and assign them to init parent
     if (vec_len(&proc->child_pcbs) > 0) {
   // retrieve the init process
164
166
       pcb_t* init_pcb =
167
            get_pcb_in_queue(&current_pcbs, 1); // init process has pid 1
168
       while (vec_len(&proc->child_pcbs) > 0) {
  pcb_t* curr_child = vec_get(&proc->child_pcbs, 0);
169
170
171
          vec_push_back(&init_pcb->child_pcbs, curr_child);
172
          vec_erase_no_deletor(&proc->child_pcbs, 0); // don't free in erase
          173
174
175
176
      }
178
179
      // decr reference counts + close files if necessary
180
      for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {</pre>
       if (proc->fd_table[i] != -1) {
181
         if (decrement_fd_ref_count(proc->fd_table[i]) == 0) {
182
           if (s_close(proc->fd_table[i]) == -1) {
183
184
              u_perror("closing on a non-valid fd");
185
186
          }
       }
187
188
189
190
      // cancel + join this thread
191
      spthread_cancel(proc->thread_handle);
192
      spthread_continue(proc->thread_handle);
193
      spthread_suspend(proc->thread_handle);
194
      spthread_join(proc->thread_handle, NULL);
195
196
      // delete this process from any queue it's in + free it
```

```
197    delete_process_from_all_queues(proc);
198    free_pcb(proc);
199 }
```

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 95 of file kern pcb.c.

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

```
135
     }
136
137
     // update parent as needed
138
     vec_push_back(&parent->child_pcbs, child);
139
140
     // add to appropriate queue
141
     put_pcb_into_correct_queue(child);
142
     vec_push_back(&current_pcbs, child);
143
144
     return child;
145 }
```

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;
   }
}

}

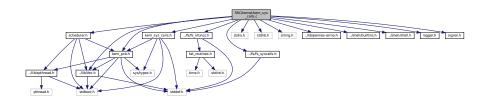
**Boar **Ind **
```

4.11 SRC/kernel/kern_sys_calls.c File Reference

```
#include "kern_sys_calls.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../fs/fs_kfuncs.h"
#include "../fs/fs_syscalls.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 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 108 of file kern_sys_calls.c.

```
108
109 int index = determine_index_in_queue(queue_to_delete_from, pid);
110 if (index != -1) {
111     vec_erase_no_deletor(queue_to_delete_from, index);
112 }
113 }
```

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 89 of file kern sys calls.c.

```
89
90  Vec* queue = NULL;
91  if (queue_id == 0) {
92    queue = &zero_priority_queue;
93  } else if (queue_id == 1) {
94    queue = &one_priority_queue;
95  } else {
96    queue = &two_priority_queue;
97  }
98
99  int index = determine_index_in_queue(queue, pid);
100  if (index != -1) {
101    vec_erase_no_deletor(queue, index);
102  }
103 }
```

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 118 of file kern_sys_calls.c.

```
119
     char* shell_argv[] = {"shell", NULL};
120
     s_spawn(shell, shell_argv, STDIN_FILENO, STDOUT_FILENO);
121
122
     // continuously wait for and reap zombie children
123
     while (true) {
     int status;
124
125
       s_waitpid(-1, &status, false);
126
128
     return NULL; // should never reach
129 }
```

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 55 of file kern_sys_calls.c.

```
58
     Vec* prev_queue;
59
    Vec* new_queue;
60
    if (prev_priority == 0) {
61
      prev_queue = &zero_priority_queue;
62
   } else if (prev_priority == 1) {
      prev_queue = &one_priority_queue;
65
66
      prev_queue = &two_priority_queue;
67
68
    if (new_priority == 0) {
70
      new_queue = &zero_priority_queue;
71
72
    } else if (new_priority == 1) {
      new_queue = &one_priority_queue;
    } else {
73
      new_queue = &two_priority_queue;
76
77
    // delete from prev_queue, if it's present at all
78
    int ind = determine_index_in_queue(prev_queue, curr_pcb->pid);
if (ind != -1) {
79
80
      vec_erase_no_deletor(prev_queue, ind);
81
83
    vec_push_back(new_queue, curr_pcb);
84 }
```

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 158 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 361 of file kern_sys_calls.c.

```
361
362
     char** argv = (char**)arg;
363
     if (argv[1] == NULL) { // no args case
364
     s_exit();
365
       return NULL;
366
367
     368
369
370
       if (s_write(current_running_pcb->output_fd, argv[i], strlen(argv[i])) ==
371
           -1) {
372
        u_perror("s_write error");
373
374
       if (s_write(current_running_pcb->output_fd, " ", 1) == -1) {
375
        u_perror("s_write error");
376
377
       i++;
378
     }
379
     if (s_write(current_running_pcb->output_fd, "\n", 1) == -1) {
380
    .=_.::ce(current_running_
u_perror("s_write error");
}
381
382
383
    return NULL;
384 }
```

4.11.1.8 s_exit()

Exits the current process and cleans up its resources.

Unconditionally exit the calling process.

Definition at line 296 of file kern_sys_calls.c.

```
296
297
      // Set process state to zombie
298
      current_running_pcb->process_state = 'Z';
299
      current_running_pcb->process_status = 20; // EXITED_NORMALLY
300
301
      // Log the exit
      log_generic_event('E', current_running_pcb->pid,
302
303
                         current_running_pcb->priority,
current_running_pcb->cmd_str);
304
305
306
     delete_from_queue(current_running_pcb->priority, current_running_pcb->pid);
307
308
     log_generic_event('Z', current_running_pcb->pid,
309
                         current_running_pcb->priority,
                         current_running_pcb->cmd_str);
310
```

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 282 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 316 of file kern sys calls.c.

```
316
317
       if (priority < 0 || priority > 2) { // error check
318
         return -1;
319
320
       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);
321
322
323
324
          log_nice_event(pid, curr_pcb->priority, priority, curr_pcb->cmd_str);
325
        curr_pcb->priority = priority;
326
         return 0;
       }
327
328
329
      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 389 of file kern_sys_calls.c.

```
char pid_top[] = "PID\tPPID\tPRI\tSTAT\tCMD\n";
390
391
     if (s_write(current_running_pcb->output_fd, pid_top, strlen(pid_top)) == -1) {
      u_perror("s_write error");
392
393
394
     for (int i = 0; i < vec_len(&current_pcbs); i++) {</pre>
395
     pcb_t* curr_pcb = (pcb_t*)vec_get(&current_pcbs, i);
396
      char buffer[100];
      397
398
399
              curr_pcb->cmd_str);
      if (s_write(current_running_pcb->output_fd, buffer, strlen(buffer)) == -1) {
401
        u_perror("s_write error");
     }
402
403
    return NULL;
404
405 }
```

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 335 of file kern_sys_calls.c.

```
336
      if (ticks <= 0) {
337
       P_ERRNO = P_EINVAL;
338
       return;
339
340
341
     // block current process, set state to sleep
     current_running_pcb->process_state = 'B';
343
     current_running_pcb->is_sleeping = true;
344
     current_running_pcb->time_to_wake = tick_counter + ticks;
     log_generic_event('B', current_running_pcb->pid,
345
346
                       current_running_pcb->priority,
347
                       current_running_pcb->cmd_str);
348
     if (spthread_suspend(current_running_pcb->thread_handle) !=
349
         0) { // give scheduler control
       perror("Error in spthread_suspend in s_sleep call");
350
351
352 }
```

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 165 of file kern_sys_calls.c.

```
165
                                                                          {
      pcb t* child:
166
      if (strcmp(argv[0], "shell") == 0) {
167
168
       child = k_proc_create(current_running_pcb, 0);
169
170
       child = k_proc_create(current_running_pcb, 1);
171
172
173
      if (child == NULL) {
       P_ERRNO = P_ENULL;
174
       return -1;
175
176
177
178
      spthread_t thread_handle;
179
180
      if (spthread_create(&thread_handle, NULL, func, argv) != 0) {
181
       perror("Error in spthread_create in s_spawn call");
182
183
184
      child->cmd_str = strdup(argv[0]);
      child->thread_handle = thread_handle;
185
186
      child->input_fd = fd0;
187
      child->output_fd = fd1;
188
      child->fd_table[0] = fd0;
189
      child->fd_table[1] = fd1;
190
191
      log_generic_event('C', child->pid, child->priority, child->cmd_str);
192
193
      return child->pid;
194 }
```

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 138 of file kern_sys_calls.c.

```
138
      pcb_t* init = k_proc_create(NULL, 0);
      if (init == NULL) {
140
       P_ERRNO = P_ENULL;
141
142
       return -1;
143
144
145
      spthread_t thread_handle;
146
      if (spthread_create(&thread_handle, NULL, init_func, NULL) != 0) {
147
       perror("Error in spthread_create in s_spawn_init call");
148
149
150
     init->cmd_str = strdup("init");
     init->thread_handle = thread_handle;
152 return init->pid;
153 }
```

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 199 of file kern_sys_calls.c.

```
199
200
       pcb_t* parent = current_running_pcb;
201
       if (parent == NULL) {
202
         return -1;
203
204
205
       // if no children, return -1
206
       bool has_child = false;
for (int i = 0; i < vec_len(&current_pcbs); i++) {</pre>
207
         pcb_t* child = vec_get(&current_pcbs, i);
208
209
          if (child->par_pid == parent->pid) {
210
            has_child = true;
211
            break;
         }
212
213
214
       if (!has_child) {
215
         return -1;
216
217
218
       \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) {
219
220
221
222
           if (wstatus != NULL) {
223
               *wstatus = child->process_status;
224
            log_generic_event('W', child->pid, child->priority, child->cmd_str);
vec_erase_no_deletor(&zombie_queue, i);
225
226
227
            delete_from_explicit_queue(&parent->child_pcbs, child->pid);
228
            k_proc_cleanup(child);
```

```
return child->pid;
230
231
232
       // If nohang is true, return immediately if no child has exited
2.3.3
234
       if (nohang) {
235
         return 0;
236
237
238
       \ensuremath{//} Block the parent until a child exits
       delete_from_queue(parent->priority, parent->pid);
parent->process_state = 'B';
239
240
       log_generic_event('B', parent->pid, parent->priority, parent->cmd_str);
241
242
243
         // 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>
244
245
246
            if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
247
248
              if (wstatus != NULL) {
249
                 *wstatus = child->process_status;
250
              \label{log_generic_event} $$\log_{\mathtt{generic_event}}('\mathtt{W}', \mathtt{child->pid}, \mathtt{child->priority}, \mathtt{child->cmd\_str})$;
2.51
              vec_erase_no_deletor(&zombie_queue, i);
delete_from_explicit_queue(&parent->child_pcbs, child->pid);
2.52
253
               k_proc_cleanup(child);
255
               return child->pid;
256
        }
2.57
258
          // scan children of current running process for non-terminated state changes for (int i = 0; i < vec_len(&parent->child_pcbs); i++) {
259
260
261
          pcb_t* child = vec_get(&parent->child_pcbs, i);
262
             if ((pid == -1 || child->pid == pid) &&
                 (child->process_status == 21 ||
263
                   child->process_status == 23)) { // signaled
264
              if (wstatus != NULL) {
265
                 *wstatus = child->process_status;
266
267
               log_generic_event('W', child->pid, child->priority, child->cmd_str);
child->process_status = 0; // reset status
268
269
               return child->pid;
2.70
271
272
         }
273
       }
274
275
       // If we get here, something went wrong
276
       return -1;
277 }
```

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.

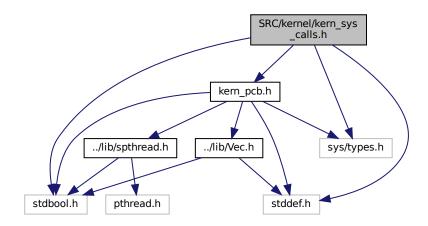
4.11.2.9 zombie_queue

```
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 108 of file kern_sys_calls.c.

```
108
109 int index = determine_index_in_queue(queue_to_delete_from, pid);
110 if (index != -1) {
    vec_erase_no_deletor(queue_to_delete_from, index);
112 }
113 }
```

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 89 of file kern sys calls.c.

```
90
     Vec* queue = NULL;
    if (queue_id == 0) {
92
      queue = &zero_priority_queue;
   } else if (queue_id == 1) {
93
94
      queue = &one_priority_queue;
   } else {
95
     queue = &two_priority_queue;
96
99
    int index = determine_index_in_queue(queue, pid);
     if (index != -1) {
100
101
       vec_erase_no_deletor(queue, index);
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.

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 118 of file kern_sys_calls.c.

```
char* shell_argv[] = {"shell", NULL};
120
      s_spawn(shell, shell_argv, STDIN_FILENO, STDOUT_FILENO);
121
     // continuously wait for and reap zombie children
122
123
     while (true) {
      int status;
124
     s_waitpid(-1, &status, false);
}
125
126
127
128 return NULL; // should never reach
129 }
```

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 55 of file kern sys calls.c.

```
58
      Vec* prev_queue;
      Vec* new_queue;
     if (prev_priority == 0) {
    prev_queue = &zero_priority_queue;
} else if (prev_priority == 1) {
  prev_queue = &one_priority_queue;
62
6.3
64
65
       prev_queue = &two_priority_queue;
67
68
     if (new_priority == 0) {
  new_queue = &zero_priority_queue;
69
70
     } else if (new_priority == 1) {
71
        new_queue = &one_priority_queue;
73
74
        new_queue = &two_priority_queue;
     }
75
76
    // delete from prev_queue, if it's present at all
int ind = determine_index_in_queue(prev_queue, curr_pcb->pid);
79
     if (ind != -1) {
80
        vec_erase_no_deletor(prev_queue, ind);
81
82
83
     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 158 of file kern_sys_calls.c.

```
158
159 k_proc_cleanup(get_pcb_in_queue(&current_pcbs, 1));
160 }
```

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 361 of file kern_sys_calls.c.

```
361
     char** argv = (char**)arg;
362
     if (argv[1] == NULL) { // no args case
363
     s_exit();
364
365
       return NULL;
     }
366
367
     368
369
370
     if (s_write(current_running_pcb->output_fd, argv[i], strlen(argv[i])) ==
371
           -1) {
372
        u_perror("s_write error");
373
      if (s_write(current_running_pcb->output_fd, " ", 1) == -1) {
  u_perror("s_write error");
374
375
376
377
378
     }
379
     if (s_write(current_running_pcb->output_fd, "\n", 1) == -1) {
380
381
      u_perror("s_write error");
    return NULL;
383
384 }
```

4.12.1.8 s_exit()

```
void s_exit (
          void )
```

Unconditionally exit the calling process.

Unconditionally exit the calling process.

Definition at line 296 of file kern sys calls.c.

```
296
297
      // Set process state to zombie
298
     current_running_pcb->process_state = 'Z';
299
     current_running_pcb->process_status = 20; // EXITED_NORMALLY
300
301
      // Log the exit
     302
303
304
                       current_running_pcb->cmd_str);
305
306
     delete_from_queue(current_running_pcb->priority, current_running_pcb->pid);
307
     log_generic_event('Z', current_running_pcb->pid,
308
309
                       current_running_pcb->priority,
current_running_pcb->cmd_str);
310
311 }
```

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 282 of file kern_sys_calls.c.

```
282
283    pcb_t* pcb_with_pid = get_pcb_in_queue(&current_pcbs, pid);
284    if (pcb_with_pid == NULL) {
285        return -1; // pid not found case
286    }
287
288    pcb_with_pid->signals[signal] = true; // signal flagged
289    log_generic_event('S', pid, pcb_with_pid->priority, pcb_with_pid->cmd_str);
290    return 0;
291 }
```

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 316 of file kern_sys_calls.c.

```
316
       if (priority < 0 || priority > 2) { // error check
318
319
320
       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);
321
322
323
324
         log_nice_event(pid, curr_pcb->priority, priority, curr_pcb->cmd_str);
325
        curr_pcb->priority = priority;
326
         return 0;
327
      }
328
329
      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 389 of file kern_sys_calls.c.

```
char pid_top[] = "PID\tPPID\tPRI\tSTAT\tCMD\n";
390
391
     if (s_write(current_running_pcb->output_fd, pid_top, strlen(pid_top)) == -1) {
       u_perror("s_write error");
392
393
394
     for (int i = 0; i < vec_len(&current_pcbs); i++) {</pre>
395
      pcb_t* curr_pcb = (pcb_t*)vec_get(&current_pcbs, i);
396
        char buffer[100];
397
       snprintf(buffer, sizeof(buffer), "%d\t%d\t%d\t%c\t%s\n", curr_pcb->pid,
398
                 curr_pcb->par_pid, curr_pcb->priority, curr_pcb->process_state,
                curr_pcb->cmd_str);
399
      if (s_write(current_running_pcb->output_fd, buffer, strlen(buffer)) == -1) {
400
401
         u_perror("s_write error");
      }
402
403
404
     return NULL;
405 }
```

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 335 of file kern_sys_calls.c.

```
340
341
     // block current process, set state to sleep
342
     current_running_pcb->process_state = 'B';
     current_running_pcb->is_sleeping = true;
343
     current_running_pcb->time_to_wake = tick_counter + ticks;
344
345
     log_generic_event('B', current_running_pcb->pid,
                       current_running_pcb->priority,
346
347
                       current_running_pcb->cmd_str);
348
     if (spthread_suspend(current_running_pcb->thread_handle) !=
349
         0) { // give scheduler control
       perror("Error in spthread_suspend in s_sleep call");
350
351
352 }
```

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 165 of file kern_sys_calls.c.

```
165
                                                                                {
166
      pcb_t* child;
167
      if (strcmp(argv[0], "shell") == 0) {
168
        child = k_proc_create(current_running_pcb, 0);
169
        child = k_proc_create(current_running_pcb, 1);
170
171
172
173
      if (child == NULL) {
174
       P_ERRNO = P_ENULL;
175
176
        return -1;
177
178
      spthread_t thread_handle;
179
180
     perror("Error in spthread_create in s_spawn call");
}
      if (spthread_create(&thread_handle, NULL, func, argv) != 0) {
181
182
183
184
      child->cmd_str = strdup(argv[0]);
185
      child->thread_handle = thread_handle;
186
      child->input_fd = fd0;
      child->output_fd = fd1;
child->fd_table[0] = fd0;
child->fd_table[1] = fd1;
187
188
189
190
191
      log_generic_event('C', child->pid, child->priority, child->cmd_str);
```

```
192
193    return child->pid;
194 }
```

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 138 of file kern_sys_calls.c.

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

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.

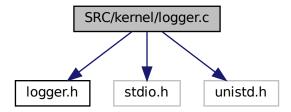
Definition at line 199 of file kern sys calls.c.

```
200
      pcb_t* parent = current_running_pcb;
      return -1;
}
201
       if (parent == NULL) {
202
203
204
205
       // if no children, return -1
      bool has_child = false;
206
207
      for (int i = 0; i < vec_len(&current_pcbs); i++) {</pre>
208
       pcb_t* child = vec_get(&current_pcbs, i);
209
        if (child->par_pid == parent->pid) {
  has_child = true;
210
211
          break;
        }
213
      if (!has_child) {
214
215
        return -1;
216
217
218
      // Scan the zombie queue first for terminated children.
219
      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) {
220
221
          if (wstatus != NULL) {
222
            *wstatus = child->process_status;
223
224
225
           log_generic_event('W', child->pid, child->priority, child->cmd_str);
226
           vec_erase_no_deletor(&zombie_queue, i);
227
           delete_from_explicit_queue(&parent->child_pcbs, child->pid);
          k_proc_cleanup(child);
228
229
           return child->pid;
230
231
232
      \ensuremath{//} If nohang is true, return immediately if no child has exited
233
      if (nohang) {
234
235
        return 0:
236
237
238
      // Block the parent until a child exits
      delete_from_queue(parent->priority, parent->pid);
parent->process_state = 'B';
239
240
      log_generic_event('B', parent->pid, parent->priority, parent->cmd_str);
241
242
243
244
        // Scan the zombie queue first for terminated children.
245
         for (int i = 0; i < vec_len(&zombie_queue); i++) {</pre>
246
           pcb_t* child = vec_get(&zombie_queue, i);
           if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
247
248
            if (wstatus != NULL) {
               *wstatus = child->process_status;
249
250
251
             log_generic_event('W', child->pid, child->priority, child->cmd_str);
252
             vec_erase_no_deletor(&zombie_queue, i);
delete_from_explicit_queue(&parent->child_pcbs, child->pid);
253
254
             k_proc_cleanup(child);
255
             return child->pid;
256
257
258
259
         // scan children of current running process for non-terminated state changes
        for (int i = 0; i < vec_len(&parent->child_pcbs); i++) {
260
          pcb_t* child = vec_get(&parent->child_pcbs, i);
261
           if ((pid == -1 || child->pid == pid) &&
262
263
               (child->process_status == 21 ||
                child->process_status == 23)) { // signaled
264
             if (wstatus != NULL) {
  *wstatus = child->process_status;
265
266
267
             log_generic_event('W', child->pid, child->priority, child->cmd_str);
child->process_status = 0; // reset status
268
269
270
             return child->pid;
271
272
        }
273
      }
```

```
275  // If we get here, something went wrong
276  return -1;
277 }
```

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 when an event is scheduled.
- void log_generic_event (char event_type, int pid, int nice_value, char *process_name)

 Logs non-nice, non-scheduling events since they have same format.

• void log_nice_event (int pid, int old_nice_value, int new_nice_value, char *process_name)

Logs a nice-related event.

4.13.1 Function Documentation

4.13.1.1 log generic event()

Logs non-nice, non-scheduling events since they have same format.

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

Definition at line 20 of file logger.c.

```
24
     char* operation;
2.5
     switch (event_type) {
  case 'C':
2.6
        operation = "CREATE";
28
29
         break;
30
      case 'S':
         operation = "SIGNALED";
31
32
         break;
      case 'E':
33
       operation = "EXITED";
break;
34
35
36
      case 'Z':
       operation = "ZOMBIE";
37
      break;
case '0':
38
39
       operation = "ORPHAN";
40
         break;
      case 'W':
       operation = "WAITED";
43
      break; case 'B':
44
4.5
       operation = "BLOCKED";
break;
46
48
       case 'U':
       operation = "UNBLOCKED";
49
      break;
case 's':
50
51
       operation = "STOPPED";
52
53
         break;
55
         operation = "CONTINUED";
56
57
    }
58
    char buffer[200];
59
    int str_len =
      snprintf(buffer, sizeof(buffer), "[%d]\t%s\t%d\t%d\t%s\n", tick_counter,
    operation, pid, nice_value, process_name);
if (write(log_fd, buffer, str_len) == -1) {
62
   perror("error in writing to the log file for generic event");
}
6.3
64
65
66 }
```

4.13.1.2 log_nice_event()

Logs a nice-related event.

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

Definition at line 71 of file logger.c.

4.13.1.3 log_scheduling_event()

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

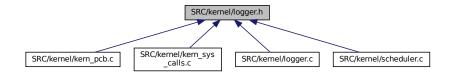
Logs when an event is scheduled.

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

Definition at line 8 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

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

Definition at line 20 of file logger.c.

```
char* operation;
   switch (event_type) {
  case 'C':
27
       operation = "CREATE";
28
     break;
case 'S':
29
30
31
       operation = "SIGNALED";
33
     case 'E':
      operation = "EXITED";
34
35
     case 'Z':
36
     operation = "ZOMBIE";
break;
case 'O':
38
39
       operation = "ORPHAN";
40
     break;
case 'W':
41
42
      operation = "WAITED";
43
45
     case 'B':
      operation = "BLOCKED";
46
     break;
case 'U':
47
48
       operation = "UNBLOCKED";
break;
       operation = "STOPPED";
52
5.3
         break;
54
     default:
55
        operation = "CONTINUED";
         break;
```

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

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

Definition at line 71 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

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

Definition at line 8 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.

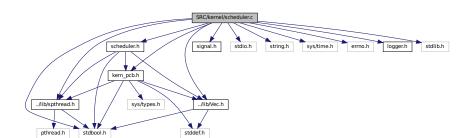
4.14.2.2 tick_counter

```
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 <stdio.h>
#include <string.h>
#include <sys/time.h>
#include "../lib/Vec.h"
#include "../lib/spthread.h"
#include "errno.h"
#include "logger.h"
#include "stdlib.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 228 of file scheduler.c.

```
228
229 tick_counter++;
230 }
```

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 201 of file scheduler.c.

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 215 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 179 of file scheduler.c.

```
179 {
180 delete_process_from_all_queues_except_current(pcb);
181 delete_process_from_particular_queue(pcb, &current_pcbs);
182 }
```

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 168 of file scheduler.c.

```
168
delete_process_from_particular_queue(pcb, &zero_priority_queue);
170
delete_process_from_particular_queue(pcb, &one_priority_queue);
171
delete_process_from_particular_queue(pcb, &two_priority_queue);
172
delete_process_from_particular_queue(pcb, &zombie_queue);
173
delete_process_from_particular_queue(pcb, &sleep_blocked_queue);
174
}
```

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 155 of file scheduler.c.

```
155
156
      for (int i = 0; i < vec_len(queue); i++) {</pre>
       pcb_t* curr_pcb = vec_get(queue, i);
157
158
        if (curr_pcb->pid == pcb->pid)
159
         vec_erase_no_deletor(queue, i);
160
          return;
161
       }
    }
162
163 }
```

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 83 of file scheduler.c.

```
// check if all queues are empty
85
     if (vec_is_empty(&zero_priority_queue) && vec_is_empty(&one_priority_queue) &&
86
          vec_is_empty(&two_priority_queue)) {
87
        return -1;
88
     int priorities_attempted = 0;
     while (priorities_attempted < 19) {
  int curr_pri = det_priorities_arr[curr_priority_arr_index];</pre>
91
92
       curr_priority_arr_index = (curr_priority_arr_index + 1) % 19;
if (curr_pri == 0 && !vec_is_empty(&zero_priority_queue)) {
9.3
94
          priorities_attempted++;
       return 0;
} else if (curr_pri == 1 && !vec_is_empty(&one_priority_queue)) {
97
98
          priorities_attempted++;
99
          return 1;
        } else if (curr_pri == 2 && !vec_is_empty(&two_priority_queue)) {
100
          priorities_attempted++;
101
102
           return 2;
103
104
105
      return -1; // should never reach
106
```

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 112 of file scheduler.c.

```
112
       if (priority == -1) { // all queues empty
113
114
        return NULL;
115
116
117
      pcb_t* next_pcb = NULL;
      if (priority == 0) {
  next_pcb = vec_get(&zero_priority_queue, 0);
118
119
        vec_erase_no_deletor(&zero_priority_queue, 0);
120
121
      } else if (priority == 1) {
122
       next_pcb = vec_get(&one_priority_queue, 0);
123
        vec_erase_no_deletor(&one_priority_queue, 0);
      } else if (priority == 2) {
  next_pcb = vec_get(&two_priority_queue, 0);
124
125
        vec_erase_no_deletor(&two_priority_queue, 0);
126
127
128
129
      return next_pcb;
130 }
```

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 187 of file scheduler.c.

4.15.1.11 handle signal()

```
void handle_signal (
    pcb_t * pcb,
    int signal )
```

Handles the specified signal for the given PCB.

Handles a signal for a given process.

Definition at line 235 of file scheduler.c.

```
235
236
       switch (signal) {
        case 0: // P SIGSTOP
237
           if (pcb->process_state == 'R' || pcb->process_state == 'B') {
238
             pcb->process_state = 'S';
log_generic_event('s', pcb->pid, pcb->priority, pcb->cmd_str);
delete_process_from_all_queues_except_current(pcb);
239
240
241
242
             pcb->process_status = 21; // STOPPED_BY_SIG
243
244
           pcb->signals[0] = false;
245
           break;
246
                                                   // P_SIGCONT
247
          if (pcb->process_state == 'S') { // Only continue if stopped
             if (pcb->is_sleeping) {
  pcb->process_state = 'B';
248
249
                delete_process_from_all_queues_except_current(pcb);
250
251
               put_pcb_into_correct_queue(pcb);
252
253
               pcb->process_state = 'R';
254
                delete_process_from_all_queues_except_current(pcb);
255
                put_pcb_into_correct_queue(pcb);
256
             log_generic_event('c', pcb->pid, pcb->priority, pcb->cmd_str);
pcb->process_status = 23; // CONT_BY_SIG
258
259
260
           pcb->signals[1] = false;
2.61
           break;
                                                   // P_SIGTERM
262
         case 2:
          if (pcb->process_state != 'Z') { // Don't terminate if already zombie
263
             pcb->process_state = 'Z';
265
             pcb->process_status = 22; // TERM_BY_SIG
              log_generic_event('Z', pcb->pid, pcb->priority, pcb->cmd_str);
266
2.67
             delete_process_from_all_queues_except_current(pcb);
268
             put_pcb_into_correct_queue(pcb);
pcb->process_status = 22; // TERM_BY_SIG
269
270
271
           pcb->signals[2] = false;
272
           break;
273
      }
274 }
```

4.15.1.12 initialize_scheduler_queues()

```
void initialize_scheduler_queues ( )
```

Initializes the scheduler queues.

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

Note

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

Definition at line 54 of file scheduler.c.

```
54

55 zero_priority_queue = vec_new(0, NULL);
56 one_priority_queue = vec_new(0, NULL);
57 two_priority_queue = vec_new(0, NULL);
58 zombie_queue = vec_new(0, NULL);
59 sleep_blocked_queue = vec_new(0, NULL);
60 current_pcbs = vec_new(0, free_pcb);
61}
```

4.15.1.13 put_pcb_into_correct_queue()

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 136 of file scheduler.c.

```
137
      if (pcb->process_state == 'R') {
138
       if (pcb->priority == 0) {
       vec_push_back(&zero_priority_queue, pcb);
} else if (pcb->priority == 1) {
139
140
141
         vec_push_back(&one_priority_queue, pcb);
       } else if (pcb->priority == 2) {
143
          vec_push_back(&two_priority_queue, pcb);
144
145
      } else if (pcb->process_state == 'Z') {
      vec_push_back(&zombie_queue, pcb);
} else if (pcb->process_state == 'B' || pcb->is_sleeping) {
146
148
        vec_push_back(&sleep_blocked_queue, pcb);
149
150 }
```

4.15.1.14 s shutdown pennos()

Shuts down the scheduler and cleans up resources.

Shuts down the PennOS scheduler.

Definition at line 279 of file scheduler.c.

```
279 {
280 scheduling_done = true;
281 }
```

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 286 of file scheduler.c.

```
286
287
      int curr_priority_queue_num;
288
289
      // mask for while scheduler is waiting for alarm
290
      sigset_t suspend_set;
291
      sigfillset(&suspend_set);
292
      sigdelset(&suspend_set, SIGALRM);
293
294
      // ensure sigarlm doesn't terminate the process
295
      struct sigaction act = (struct sigaction) {
296
          .sa_handler = alarm_handler,
297
           .sa_mask = suspend_set,
298
           .sa_flags = SA_RESTART,
299
300
      sigaction (SIGALRM, &act, NULL);
301
302
      // make sure SIGALRM is unblocked
303
      sigset_t alarm_set;
304
      sigemptyset(&alarm_set);
305
      sigaddset (&alarm set, SIGALRM);
      pthread_sigmask(SIG_UNBLOCK, &alarm_set, NULL);
306
308
      struct itimerval it;
309
      it.it_interval = (struct timeval) {.tv_usec = hundred_millisec};
310
      it.it_value = it.it_interval;
      setitimer(ITIMER_REAL, &it, NULL);
311
312
313
      while (!scheduling_done) {
        // handle signals for the currently running process
315
        if (current_running_pcb != NULL) {
316
           for (int i = 0; i < 3; i++) {
             if (current_running_pcb->signals[i]) {
317
              handle_signal(current_running_pcb, i);
318
               // If process was terminated, don't continue scheduling it
319
320
               if (current_running_pcb->process_state != 'R') {
321
                 current_running_pcb = NULL;
322
                 break;
323
324
            }
325
          }
326
327
328
        // 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>
329
330
          for (int j = 0; j < 3; j++) {
   if (curr_pcb->signals[j]) {
331
332
333
              handle_signal(curr_pcb, j);
334
335
          }
336
337
338
        // Check sleep/blocked queue to move processes back to scheduable queues
339
        for (int i = 0; i < vec_len(&sleep_blocked_queue); i++) {</pre>
340
          pcb_t* blocked_proc = vec_get(&sleep_blocked_queue, i);
          bool make_runnable = false;
if (blocked_proc->is_sleeping &&
341
342
               blocked_proc->time_to_wake <= tick_counter) {
343
             blocked_proc->is_sleeping = false;
344
345
             blocked_proc->time_to_wake = -1;
346
            blocked_proc->signals[2] = false; // Unlikely, but reset signal
347
            make_runnable = true;
348
          } else if (blocked_proc->is_sleeping &&
                      blocked_proc->signals[2]) { // P_SIGTERM received
349
350
             blocked_proc->is_sleeping = false;
351
             blocked_proc->process_state = 'Z';
352
             blocked_proc->process_status = 22; // TERM_BY_SIG
353
             blocked_proc->signals[2] = false;
354
            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,
355
356
                                blocked_proc->cmd_str);
```

```
i--;
358
359
          } else if (child_in_zombie_queue(blocked_proc)) {
360
            make_runnable = true;
          } else if (child_with_changed_process_status(blocked_proc)) {
361
362
            make_runnable = true;
363
364
365
          if (make_runnable) {
366
           blocked_proc->process_state = 'R';
367
            vec_erase_no_deletor(&sleep_blocked_queue, i);
            delete_process_from_all_queues_except_current(blocked_proc);
368
            put_pcb_into_correct_queue(blocked_proc);
log_generic_event('U', blocked_proc->pid, blocked_proc->priority,
369
370
371
                                blocked_proc->cmd_str);
372
373
374
375
376
        curr_priority_queue_num = generate_next_priority();
377
378
        current_running_pcb = get_next_pcb(curr_priority_queue_num);
379
        if (current_running_pcb == NULL) {
380
          sigsuspend(&suspend_set); // idle until signal received
381
          continue;
382
383
384
        log_scheduling_event(current_running_pcb->pid, curr_priority_queue_num,
385
                              current_running_pcb->cmd_str);
386
387
        if (spthread_continue(current_running_pcb->thread_handle) != 0 &&
388
            errno != EINTR) {
389
          perror("spthread_continue failed in scheduler");
390
391
        sigsuspend(&suspend_set);
        if (spthread_suspend(current_running_pcb->thread_handle) != 0 &&
    errno != EINTR) {
392
393
394
          perror("spthread_suspend failed in scheduler");
395
396
        put_pcb_into_correct_queue(current_running_pcb);
397
398 }
```

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

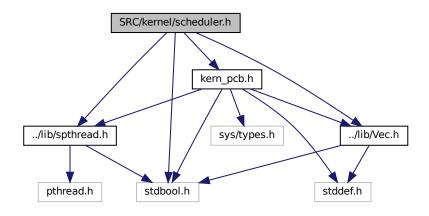
```
Vec zombie_queue
```

Definition at line 27 of file scheduler.c.

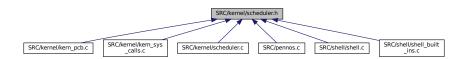
4.16 SRC/kernel/scheduler.h File Reference

```
#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

signum The signal number (unused in this implementation	on).
---	------

Handles the alarm signal.

Definition at line 228 of file scheduler.c.

```
228
229 tick_counter++;
230 }
```

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 201 of file scheduler.c.

```
201
202     for (int i = 0; i < vec_len(&zombie_queue); i++) {
203         pcb_t* child = vec_get(&zombie_queue, i);
204         if (child->par_pid == parent->pid) {
205             return true;
206         }
207     }
208     return false;
209 }
```

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 215 of file scheduler.c.

4.16.1.4 delete process from all queues()

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

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

Parameters

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

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

Definition at line 179 of file scheduler.c.

```
179 {
180 delete_process_from_all_queues_except_current(pcb);
181 delete_process_from_particular_queue(pcb, &current_pcbs);
182 }
```

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 168 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);
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 155 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 83 of file scheduler.c.

```
// check if all queues are empty
84
85
                      \begin{tabular}{ll} if (vec\_is\_empty(\&zero\_priority\_queue) &\& vec\_is\_empty(\&one\_priority\_queue) &\& vec\_is\_empty(\&one\_pr
                                       vec_is_empty(&two_priority_queue)) {
86
                              return -1;
88
89
90
                     int priorities_attempted = 0;
                     while (priorities_attempted < 19) {</pre>
91
                             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)) {
93
95
                               priorities_attempted++;
96
                                        return 0;
                             } else if (curr_pri == 1 && !vec_is_empty(&one_priority_queue)) {
97
98
                                    priorities_attempted++;
99
                                       return 1;
                                 } else if (curr_pri == 2 && !vec_is_empty(&two_priority_queue)) {
100
101
                                          priorities_attempted++;
102
                                            return 2;
103
104
                        }
105
                        return -1; // should never reach
107 }
```

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 112 of file scheduler.c.

```
112
       if (priority == -1) { // all queues empty
113
         return NULL;
114
115
116
117
       pcb_t* next_pcb = NULL;
       if (priority == 0) {
  next_pcb = vec_get(&zero_priority_queue, 0);
  vec_erase_no_deletor(&zero_priority_queue, 0);
118
119
120
121
       } else if (priority == 1) {
        next_pcb = vec_get(&one_priority_queue, 0);
         vec_erase_no_deletor(&one_priority_queue, 0);
123
124
       } else if (priority == 2) {
       next_pcb = vec_get(&two_priority_queue, 0);
vec_erase_no_deletor(&two_priority_queue, 0);
125
126
127
128
129
       return next_pcb;
130 }
```

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 187 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 235 of file scheduler.c.

```
236
        switch (signal) {
237
          case 0: // P_SIGSTOP
            ase 0: // P_SIGSTOP
if (pcb->process_state == 'R' || pcb->process_state == 'B') {
   pcb->process_state = 'S';
   log_generic_event('s', pcb->pid, pcb->priority, pcb->cmd_str);
   delete_process_from_all_queues_except_current(pcb);
238
239
240
242
               pcb->process_status = 21; // STOPPED_BY_SIG
243
244
            pcb->signals[0] = false;
245
             break;
246
                                                           // P_SIGCONT
          case 1:
            if (pcb->process_state == 'S') { // Only continue if stopped
              if (pcb->is_sleeping) {
  pcb->process_state = 'B';
249
250
                  delete_process_from_all_queues_except_current(pcb);
251
                  put_pcb_into_correct_queue(pcb);
252
             } else {
               pcb->process_state = 'R';
253
254
                  delete_process_from_all_queues_except_current(pcb);
255
                  put_pcb_into_correct_queue(pcb);
256
               log_generic_event('c', pcb->pid, pcb->priority, pcb->cmd_str);
pcb->process_status = 23; // CONT_BY_SIG
257
258
259
             pcb->signals[1] = false;
261
             break;
262
          case 2:
                                                           // P_SIGTERM
           if (pcb->process_state != 'Z') { // Don't terminate if already zombie
  pcb->process_state = 'Z';
263
2.64
               pcb->process_status = 22; // TERM_BY_SIG
265
               log_generic_event('Z', pcb->pid, pcb->priority, pcb->cmd_str);
delete_process_from_all_queues_except_current(pcb);
266
               put_pcb_into_correct_queue(pcb);
pcb->process_status = 22; // TERM_BY_SIG
268
269
270
271
             pcb->signals[2] = false;
272
             break;
273
274 }
```

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 gueues. 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 136 of file scheduler.c.

```
136
137
      if (pcb->process_state == 'R') {
138
       if (pcb->priority == 0) {
139
          vec_push_back(&zero_priority_queue, pcb);
140
        } else if (pcb->priority == 1) {
          vec_push_back(&one_priority_queue, pcb);
141
       } else if (pcb->priority == 2) {
142
143
          vec_push_back(&two_priority_queue, pcb);
144
145
      } else if (pcb->process_state == 'Z') {
     vec_push_back(&zombie_queue, pcb);
} else if (pcb->process_state == 'B' || pcb->is_sleeping) {
146
147
        vec_push_back(&sleep_blocked_queue, pcb);
148
149
150 }
```

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 279 of file scheduler.c.

```
279 {
280 scheduling_done = true;
281 }
```

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 286 of file scheduler.c.

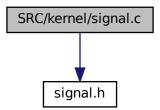
```
286
287
      int curr_priority_queue_num;
288
289
      // mask for while scheduler is waiting for alarm
290
      sigset_t suspend_set;
291
      sigfillset(&suspend_set);
292
      sigdelset (&suspend_set, SIGALRM);
293
294
      // ensure sigarlm doesn't terminate the process
295
      struct sigaction act = (struct sigaction) {
296
          .sa_handler = alarm_handler,
297
           .sa_mask = suspend_set,
298
           .sa_flags = SA_RESTART,
299
300
      sigaction(SIGALRM, &act, NULL);
301
302
      // make sure SIGALRM is unblocked
303
      sigset_t alarm_set;
304
      sigemptyset(&alarm_set);
305
      sigaddset (&alarm set, SIGALRM);
      pthread_sigmask(SIG_UNBLOCK, &alarm_set, NULL);
306
308
      struct itimerval it;
309
      it.it_interval = (struct timeval) {.tv_usec = hundred_millisec};
310
      it.it_value = it.it_interval;
      setitimer(ITIMER_REAL, &it, NULL);
311
312
313
      while (!scheduling_done) {
        // handle signals for the currently running process
315
        if (current_running_pcb != NULL) {
316
           for (int i = 0; i < 3; i++) {
             if (current_running_pcb->signals[i]) {
317
              handle_signal(current_running_pcb, i);
318
               // If process was terminated, don't continue scheduling it
319
320
               if (current_running_pcb->process_state != 'R') {
321
                 current_running_pcb = NULL;
322
                 break;
323
324
            }
325
          }
326
327
328
        // 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>
329
330
          for (int j = 0; j < 3; j++) {
   if (curr_pcb->signals[j]) {
331
332
333
              handle_signal(curr_pcb, j);
334
335
          }
336
337
338
        // Check sleep/blocked queue to move processes back to scheduable queues
339
        for (int i = 0; i < vec_len(&sleep_blocked_queue); i++) {</pre>
340
          pcb_t* blocked_proc = vec_get(&sleep_blocked_queue, i);
          bool make_runnable = false;
if (blocked_proc->is_sleeping &&
341
342
               blocked_proc->time_to_wake <= tick_counter) {
343
             blocked_proc->is_sleeping = false;
344
345
             blocked_proc->time_to_wake = -1;
346
            blocked_proc->signals[2] = false; // Unlikely, but reset signal
347
            make_runnable = true;
348
          } else if (blocked_proc->is_sleeping &&
                      blocked_proc->signals[2]) { // P_SIGTERM received
349
350
             blocked_proc->is_sleeping = false;
351
             blocked_proc->process_state = 'Z';
352
             blocked_proc->process_status = 22; // TERM_BY_SIG
353
             blocked_proc->signals[2] = false;
354
            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,
355
356
                                blocked_proc->cmd_str);
```

```
i--;
359
          } else if (child_in_zombie_queue(blocked_proc)) {
360
            make_runnable = true;
          } else if (child_with_changed_process_status(blocked_proc)) {
361
362
            make_runnable = true;
363
364
365
          if (make_runnable) {
366
            blocked_proc->process_state = 'R';
367
            vec_erase_no_deletor(&sleep_blocked_queue, i);
            delete_process_from_all_queues_except_current(blocked_proc);
368
            put_pcb_into_correct_queue(blocked_proc);
log_generic_event('U', blocked_proc->pid, blocked_proc->priority,
369
370
371
                                blocked_proc->cmd_str);
372
373
374
375
376
        curr_priority_queue_num = generate_next_priority();
377
378
        current_running_pcb = get_next_pcb(curr_priority_queue_num);
379
        if (current_running_pcb == NULL) {
380
          sigsuspend(&suspend_set); // idle until signal received
381
          continue;
382
383
384
        log_scheduling_event(current_running_pcb->pid, curr_priority_queue_num,
385
                               current_running_pcb->cmd_str);
386
387
        if (spthread_continue(current_running_pcb->thread_handle) != 0 &&
388
            errno != EINTR) {
389
          perror("spthread_continue failed in scheduler");
390
391
        sigsuspend(&suspend_set);
        if (spthread_suspend(current_running_pcb->thread_handle) != 0 &&
    errno != EINTR) {
392
393
394
          perror("spthread_suspend failed in scheduler");
395
396
        put_pcb_into_correct_queue(current_running_pcb);
397
398 }
```

4.17 SRC/kernel/signal.c File Reference

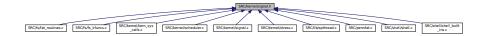
#include <signal.h>

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.

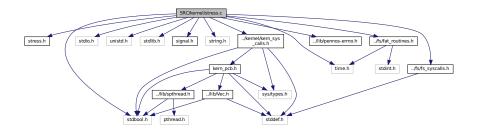
4.18.1.10 TERM_BY_SIG

```
#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()

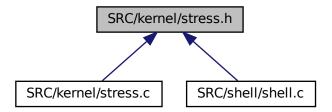
```
void* nohang (
     void * arg )
```

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 }
```

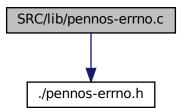
4.20.1.4 recur()

```
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

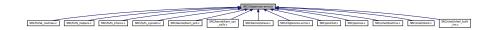
4.21.1.1 P_ERRNO

```
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 9 of file pennos-errno.h.

4.22.1.2 P_EBUSY

```
#define P_EBUSY 6
```

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

4.22.1.3 P_ECLOSE

```
#define P_ECLOSE 19
```

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

4.22.1.4 P_ECOMMAND

```
#define P_ECOMMAND 21
```

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

4.22.1.5 P_EEXIST

```
#define P_EEXIST 5
```

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

4.22.1.6 P_EFS_NOT_MOUNTED

```
#define P_EFS_NOT_MOUNTED 8
```

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

4.22.1.7 P_EFULL

```
#define P_EFULL 7
```

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

4.22.1.8 P_EFUNC

```
#define P_EFUNC 14
```

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

4.22.1.9 P_EINTR

```
#define P_EINTR 9
```

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

4.22.1.10 P EINVAL

```
#define P_EINVAL 4
```

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

4.22.1.11 P_ELSEEK

```
#define P_ELSEEK 12
```

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

4.22.1.12 P_EMALLOC

```
#define P_EMALLOC 16
```

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

4.22.1.13 P_EMAP

```
#define P_EMAP 13
```

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

4.22.1.14 P_ENOENT

```
#define P_ENOENT 1
```

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

4.22.1.15 P_ENULL

```
#define P_ENULL 10
```

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

4.22.1.16 P EOPEN

```
#define P_EOPEN 15
```

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

4.22.1.17 P_EPARSE

#define P_EPARSE 20

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

4.22.1.18 P_EPERM

```
#define P_EPERM 3
```

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

4.22.1.19 P_EREAD

```
#define P_EREAD 11
```

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

4.22.1.20 P_EREDIR

```
#define P_EREDIR 24
```

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

4.22.1.21 P_ESIGNAL

```
#define P_ESIGNAL 17
```

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

4.22.1.22 P EUNKNOWN

```
#define P_EUNKNOWN 99
```

Definition at line 32 of file pennos-errno.h.

4.22.1.23 P_EWRITE

```
#define P_EWRITE 18
```

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

4.22.1.24 P_INITFAIL

```
#define P_INITFAIL 23
```

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

4.22.1.25 P_NEEDF

```
#define P_NEEDF 22
```

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

4.22.2 Variable Documentation

4.22.2.1 P_ERRNO

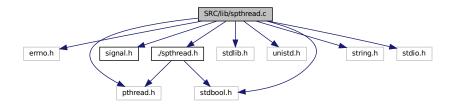
```
int P_ERRNO [extern]
```

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.

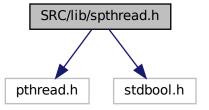
```
171
172
      pthread_t pself = pthread_self();
173
174
      if (pthread_equal(pself, thread.thread) != 0) {
175
        return spthread_suspend_self();
176
177
178
      spthread_signal_args args = (spthread_signal_args) {
179
          .signal = SPTHREAD_SIG_SUSPEND,
          .ack = 0,
180
181
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()

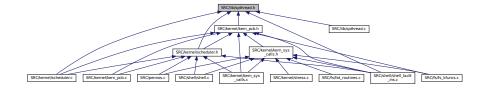
```
int spthread\_suspend\_self ( )
Definition at line 225 of file spthread.c.
        spthread_t self;
226
       bool am_sp = spthread_self(&self);
if (!am_sp) {
  return ESRCH;
227
228
229
230
231
       my_meta->state = SPTHREAD_SUSPENDED_STATE;
233
234
       sigsuspend(&my_meta->suspend_set);
} 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
2.78
        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()

```
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()

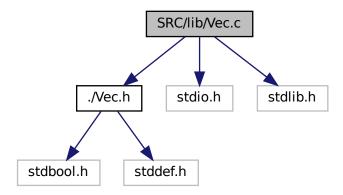
```
int spthread_suspend (
                spthread_t thread )
Definition at line 171 of file spthread.c.
171
172
      pthread_t pself = pthread_self();
173
174
      if (pthread_equal(pself, thread.thread) != 0) {
      return spthread_suspend_self();
}
175
176
177
      spthread_signal_args args = (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

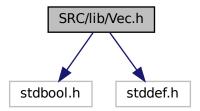
Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

Definition at line 142 of file Vec.c.

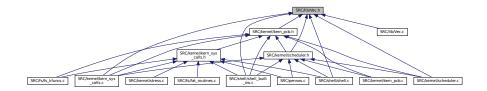
```
142
143     if (index >= self->length) {
144         perror("vec_set: idx >= len");
145     }
146     if (self->ele_dtor_fn) {
147         self->ele_dtor_fn(self->data[index]);
148     }
149     self->data[index] = new_ele;
150 }
```

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

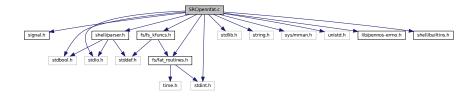
Assumes self points to a valid vector. If the index is >= self->length then this function will call perror

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 <signal.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>
#include "fs/fat_routines.h"
#include "fs/fs_kfuncs.h"
#include "lib/pennos-errno.h"
#include "shell/builtins.h"
#include "shell/parser.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

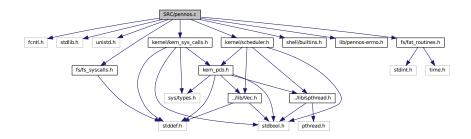
4.27.2.1 main()

```
int main (
                int argc,
                char * argv[] )
Definition at line 24 of file pennfat.c.
25
     // register signal handlers
26
     struct sigaction sa;
     sa.sa_handler = signal_handler;
28
     sigemptyset(&sa.sa_mask);
29
     sa.sa_flags = SA_RESTART;
30
     // set up handler for SIGINT (ctrl-c)
     if (sigaction(SIGINT, &sa, NULL) == -1) {
32
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
       // print prompt
if (k_write(STDOUT_FILENO, PROMPT, strlen(PROMPT)) < 0) {</pre>
48
49
         P_ERRNO = P_EWRITE;
         u_perror("prompt write error");
51
52
         break;
       }
53
54
55
       // read user input
56
       int bytes_read =
           k_read(STDIN_FILENO, input_buffer, sizeof(input_buffer) - 1);
58
59
       // check for EOF (ctrl-D) \,
       if (bytes_read <= 0) {</pre>
60
        k_write(STDOUT_FILENO, "\n", 1);
61
62
         break;
63
64
       // remove trailing newline if present
if (bytes_read > 0 && input_buffer[bytes_read - 1] == '\n') {
6.5
66
67
         input_buffer[bytes_read - 1] = '\0';
68
69
70
       // parse command and check error
71
       struct parsed_command* parsed_command = NULL;
72
       int parse_result = parse_command(input_buffer, &parsed_command);
       if (parse_result != 0) {
73
         if (parse_result == -1) {
75
          P_ERRNO = P_EINVAL;
76
           u_perror("Error parsing command");
77
         } else {
78
           print_parser_errcode(stderr, parse_result);
         }
79
80
         continue:
81
       }
82
83
       // skip empty commands
84
       if (parsed_command->num_commands == 0) {
85
        free (parsed_command);
86
         continue;
87
88
89
       \ensuremath{//} extract command and arguments
90
       char** args = parsed_command->commands[0];
91
92
       // execute command
       if (strcmp(args[0], "mkfs") == 0) {
93
         if (args[1] == NULL || args[2] == NULL || args[3] == NULL) {
   P_ERRNO = P_EINVAL;
95
           u_perror("mkfs");
96
97
         } else {
98
           int blocks in fat = atoi(args[2]);
           int block_size = atoi(args[3]);
99
100
            if (mkfs(args[1], blocks_in_fat, block_size) != 0) {
```

```
101
              u_perror("mkfs");
102
103
104
        } else if (strcmp(args[0], "mount") == 0) {
          if (args[1] == NULL) {
  P_ERRNO = P_EINVAL;
106
            u_perror("mount");
107
108
109
            if (mount(args[1]) != 0) {
110
              u_perror("mount");
            }
111
112
113
        } else if (strcmp(args[0], "unmount") == 0) {
114
          if (unmount() != 0) {
115
            u_perror("unmount");
116
        } else if (strcmp(args[0], "ls") == 0) {
117
        ls(args);
} else if (strcmp(args[0], "touch") == 0) {
118
119
          touch (args);
121
        } else if (strcmp(args[0], "cat") == 0) {
122
          cat (args);
        } else if (strcmp(args[0], "chmod") == 0) {
123
124
         chmod(args);
125
        } else if (strcmp(args[0], "mv") == 0) {
126
         mv(args);
127
        } else if (strcmp(args[0], "rm") == 0) {
128
        } else if (strcmp(args[0], "cp") == 0) {
129
130
         cp (args);
        } else if (strcmp(args[0], "cmpctdir") == 0) { // extra credit
131
132
          cmpctdir(args);
133
134
          P_ERRNO = P_ECOMMAND;
          u_perror("shell");
135
136
137
138
        free(parsed_command);
139
      return EXIT_SUCCESS;
141 }
```

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.

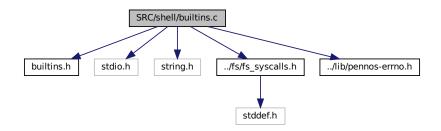
4.28.2.2 tick_counter

```
int tick_counter [extern]
```

Definition at line 35 of file scheduler.c.

4.29 SRC/shell/builtins.c File Reference

```
#include "builtins.h"
#include <stdio.h>
#include <string.h>
#include "../fs/fs_syscalls.h"
#include "../lib/pennos-errno.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.

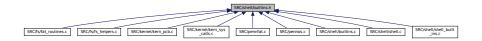
Definition at line 15 of file builtins.c.

```
16
     char buffer[256];
17
     const char *error_msg;
18
     switch (P_ERRNO) {
19
      case P_ENOENT:
20
         error_msg = "file does not exist";
21
22
         break;
2.3
       case P EBADF:
        error_msg = "bad file descriptor";
24
25
         break:
       case P_EPERM:
26
        error_msg = "operation not permitted";
27
28
         break;
       case P_EINVAL:
   error_msg = "invalid arg";
29
30
31
         break;
       case P_EEXIST:
32
       error_msg = "file already exists";
33
34
       case P_EBUSY:
35
        error_msg = "file is busy or open";
36
37
         break;
38
       case P_EFULL:
        error_msg = "no space left on device";
39
40
41
       case P_EINTR:
         error_msg = "interrupted system call";
42
43
         break:
       case P_ENULL:
44
45
        error_msg = "NULL returned unexpectedly";
46
         break;
47
       case P_EUNKNOWN:
48
         error_msg = "unknown error";
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:
       error_msg = "interrupted system call";
60
61
         break;
       case P_EOPEN:
62
        error_msg = "interrupted open call";
63
         break;
65
       case P_EMALLOC:
         error_msg = "error when trying to malloc";
66
67
       case P_EFS_NOT_MOUNTED:
68
        error_msg = "file system not mounted yet";
69
70
         break;
71
       case P_ESIGNAL:
72
         error_msg = "error with signal handling";
         break;
73
       case P EWRITE:
74
        error_msg = "interrupted write call";
75
76
         break;
       case P_ECLOSE:
    error_msg = "interrupted close call";
78
79
         break;
       case P_EPARSE:
   error_msg = "error when trying to parse a command";
80
81
82
         break:
       case P_ECOMMAND:
83
84
        error_msg = "command not found";
85
         break;
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";
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) {
       perror("s_write");
99
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.

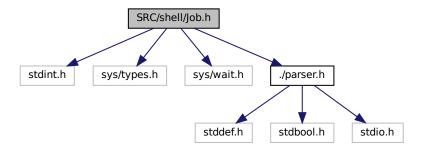
```
16
     char buffer[256];
17
     const char *error msg:
19
   switch (P_ERRNO) {
     case P_ENOENT:
20
        error_msg = "file does not exist";
21
22
         break;
      case P_EBADF:
23
       error_msg = "bad file descriptor";
         break;
26
27
      case P_EPERM:
       error_msg = "operation not permitted";
      break;
case P_EINVAL:
  error_msg = "invalid arg";
  break;
28
29
30
31
      case P_EEXIST:
    error_msg = "file already exists";
33
34
         break;
      case P_EBUSY:
35
        error_msg = "file is busy or open";
36
         break;
```

```
case P_EFULL:
38
        error_msg = "no space left on device";
40
       case P EINTR:
41
       error_msg = "interrupted system call";
42
43
         break:
      case P_ENULL:
45
       error_msg = "NULL returned unexpectedly";
46
       case P_EUNKNOWN:
47
       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";
53
54
55
         break;
       case P_EMAP:
56
       error_msg = "interrupted mmap/munmap call";
       case P_EFUNC:
59
       error_msg = "interrupted system call";
60
61
         break;
       case P_EOPEN:
62
        error_msg = "interrupted open call";
63
64
      case P_EMALLOC:
    error_msg = "error when trying to malloc";
65
66
67
         break:
       case P_EFS_NOT_MOUNTED:
68
       error_msg = "file system not mounted yet";
69
70
71
       case P_ESIGNAL:
72
       error_msg = "error with signal handling";
73
         break;
      case P_EWRITE:
    error_msg = "interrupted write call";
74
75
         break;
       case P_ECLOSE:
  error_msg = "interrupted close call";
77
78
79
         break:
       case P_EPARSE:
       error_msg = "error when trying to parse a command"; break;
80
81
83
       case P_ECOMMAND:
84
       error_msg = "command not found";
8.5
         break;
      case P_NEEDF:
86
        error_msg = "no file provided to mount";
87
88
         break;
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
```

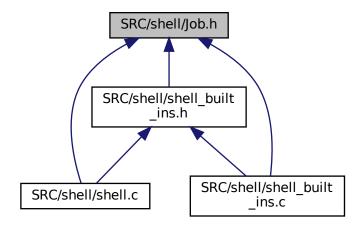
4.31 SRC/shell/Job.h File Reference

```
#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	

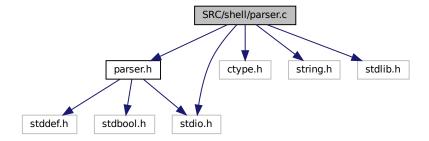
Definition at line 13 of file Job.h.

```
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 }
```

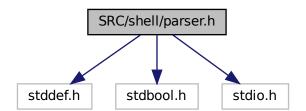
4.32.2.3 print_parser_errcode()

```
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 }
```

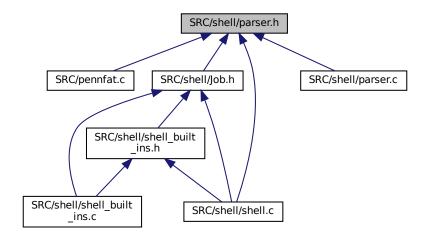
4.33 SRC/shell/parser.h File Reference

```
#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()

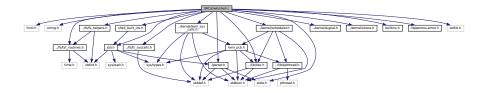
```
void print_parser_errcode (
               FILE * output,
               int err_code )
Definition at line 230 of file parser.c.
230
      switch (err_code) {
       case UNEXPECTED_FILE_INPUT:
233
          fprintf(output, "UNEXPECTED INPUT REDIRECTION TO A FILE\n");
234
        case UNEXPECTED_FILE_OUTPUT:
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 <fcntl.h>
#include <string.h>
#include "../fs/fat_routines.h"
#include "../fs/fs_helpers.h"
#include "../fs/fs_syscalls.h"
#include "../kernel/kern_sys_calls.h"
#include "../kernel/scheduler.h"
#include "../kernel/signal.h"
#include "../kernel/stress.h"
#include "../lib/Vec.h"
#include "Job.h"
#include "builtins.h"
#include "lib/pennos-errno.h"
#include "parser.h"
#include "shell_built_ins.h"
#include "stdio.h"
```

#include "stdlib.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 29 of file shell.c.

4.34.1.2 MAX_LINE_BUFFER_SIZE

```
#define MAX_LINE_BUFFER_SIZE 128
```

Definition at line 30 of file shell.c.

4.34.1.3 PROMPT

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

Definition at line 26 of file shell.c.

4.34.2 Function Documentation

4.34.2.1 execute command()

Helper function to execute a parsed command from the shell. In particular, it spawns a child process to execute the command if the built-in should run as a separate process. Otherwise, it just calls the subroutine directly.

Parameters

```
cmd the parsed command to execute, assumed non-null
```

Returns

the created child id on successful spawn, 0 on successful subroutine call, -1 when nothing was called

Definition at line 266 of file shell.c.

```
266
267  // setup fds
268  int input_fd = STDIN_FILENO; // standard fds
269  int output_fd = STDOUT_FILENO;
270
271  if (cmd->stdin_file != NULL) {
```

```
input_fd = s_open(cmd->stdin_file, F_READ);
273
          if (input_fd < 0) {</pre>
             input_fd = STDIN_FILENO; // reset to default
274
275
276
277
278
        if (cmd->is_file_append) {
279
          output_fd = s_open(cmd->stdout_file, F_APPEND);
          is_append = 1;
280
        } else {
281
          output_fd = s_open(cmd->stdout_file, F_WRITE);
282
          is_append = 0;
283
284
285
        if (output_fd < 0) {</pre>
286
          output_fd = STDOUT_FILENO; // reset to default
287
288
289
        // check for independently scheduled processes
        if (strcmp(cmd->commands[0][0], "cat") == 0) {
290
          return s_spawn(u_cat, cmd->commands[0], input_fd, output_fd);
291
292
        } else if (strcmp(cmd->commands[0][0], "sleep") == 0) {
293
          return s_spawn(u_sleep, cmd->commands[0], input_fd, output_fd);
       } else if (strcmp(cmd->commands[0][0], "busy") == 0) {
294
       return s_spawn(u_busy, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "echo") == 0) {
  return s_spawn(u_echo, cmd->commands[0], input_fd, output_fd);
295
296
297
298
        } else if (strcmp(cmd->commands[0][0], "ls") == 0) {
       return s_spawn(u_ls, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "touch") == 0) {
299
300
       return s_spawn(u_touch, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "mv") == 0) {
301
302
       return s_spawn(u_mv, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "cp") == 0) {
303
304
       return s_spawn(u_cp, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "rm") == 0) {
   return s_spawn(u_rm, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "chmod") == 0) {
   return s_spawn(u_chmod, cmd->commands[0], input_fd, output_fd);
305
306
307
308
309
310
       } else if (strcmp(cmd->commands[0][0], "ps") == 0) {
       return s_spawn(u_ps, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "kill") == 0) {
311
312
          return s_spawn(u_kill, cmd->commands[0], input_fd, output_fd);
313
       } else if (strcmp(cmd->commands[0][0]], "zombify") == 0) {
   return s_spawn(u_zombify, cmd->commands[0], input_fd, output_fd);
314
315
        } else if (strcmp(cmd->commands[0][0], "orphanify") == 0) {
316
317
          return s_spawn(u_orphanify, cmd->commands[0], input_fd, output_fd);
318
       } else if (strcmp(cmd->commands[0][0], "hang") == 0) {
       return s_spawn(hang, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "nohang") == 0) {
  return s_spawn(nohang, cmd->commands[0], input_fd, output_fd);
} else if (strcmp(cmd->commands[0][0], "recur") == 0) {
319
320
321
322
323
          return s_spawn(recur, cmd->commands[0], input_fd, output_fd);
324
        } else if (strcmp(cmd->commands[0][0], "crash") == 0) {
325
          return s_spawn(crash, cmd->commands[0], input_fd, output_fd);
326
327
328
        // check for sub-routines
        if (strcmp(cmd->commands[0][0], "nice") == 0) {
329
330
          u_nice(cmd->commands[0]);
331
          return 0;
        } else if (strcmp(cmd->commands[0][0], "nice_pid") == 0) {
332
333
          u nice pid(cmd->commands[0]);
334
          return 0;
        } else if (strcmp(cmd->commands[0][0], "man") == 0) {
335
336
          u_man(cmd->commands[0]);
337
          return 0;
338
        } else if (strcmp(cmd->commands[0][0], "bg") == 0) {
          u_bg(cmd->commands[0]);
339
340
          return 0:
        } else if (strcmp(cmd->commands[0][0], "fg") == 0) {
341
342
          u_fg(cmd->commands[0]);
343
          return 0;
344
        } else if (strcmp(cmd->commands[0][0], "jobs") == 0) {
345
          u_jobs(cmd->commands[0]);
346
          return 0;
347
        } else if (strcmp(cmd->commands[0][0], "logout") == 0) {
348
          u_logout(cmd->commands[0]);
349
          return 0;
350
351
        // otherwise, try to run command as a script
352
353
        int script_fd_open = s_open(cmd->commands[0][0], F_READ);
        if (script_fd_open < 0) { // if not a file, just move on</pre>
354
355
356
        if (has_executable_permission(script_fd_open) != 1) {
357
358
          if (s close(script fd open) == -1) {
```

```
u_perror("s_close error i.e. not a valid fd");
360
361
        return -1;
362
      } else {
363
        script_fd = script_fd_open; // update global
        input_fd_script = input_fd;
output_fd_script = output_fd;
364
365
366
367
        char* script_argv[] = {cmd->commands[0][0], NULL};
368
        pid_t wait_on =
369
             s_spawn(u_read_and_execute_script, script_argv, input_fd, output_fd);
370
        int status:
        s_waitpid(wait_on, &status, false); // wait for script to finish
371
372
        script_fd = -1;
                                                  // reset global
        input_fd_script = STDIN_FILENO;
output_fd_script = STDOUT_FILENO;
373
374
375
        if (s_close(script_fd_open) == -1) {
376
          u_perror("s_close error i.e. not a valid fd");
377
378
        return 0;
379
      }
380
     return -1; // no matches case
381
382 1
```

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
buffe	the buffer to fill with characters

Definition at line 120 of file shell.c.

```
120
                                                                        {
121
      int i = 0;
122
      char currChar;
123
      while (i < MAX_LINE_BUFFER_SIZE - 1) {</pre>
124
        int bytes_read = s_read(fd, &currChar, 1);
125
        if (bytes_read <= 0 || currChar == '\n') { // EOF or newline cases</pre>
126
127
128
        buffer[i] = currChar;
129
130
131
     buffer[i] = ' \setminus 0'; // Null-terminate the string, replaces \setminus n
132 }
```

4.34.2.3 free_job_ptr()

4.34.2.4 setup_terminal_signal_handlers()

```
void setup_terminal_signal_handlers (
                  void )
Definition at line 74 of file shell.c.
      struct sigaction sa_int = {0};
sa_int.sa_handler = shell_sigint_handler;
75
76
      sigemptyset(&sa_int.sa_mask);
78
      sa_int.sa_flags = SA_RESTART;
      if (sigaction(SIGINT, &sa_int, NULL) == -1) {
79
        P_ERRNO = P_ESIGNAL;
u_perror("sigaction");
80
81
        exit (EXIT_FAILURE);
83
84
     struct sigaction sa_stp = {0};
sa_stp.sa_handler = shell_sigstp_handler;
85
86
     sigemptyset(&sa_stp.sa_mask);
sa_stp.sa_flags = SA_RESTART;
      if (sigaction(SIGTSTP, &sa_stp, NULL) == -1) {
        P_ERRNO = P_ESIGNAL;
u_perror("sigaction");
90
91
92
        exit(EXIT_FAILURE);
93
```

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 388 of file shell.c.

```
388
389
        job_list = vec_new(0, free_job_ptr);
390
391
        setup_terminal_signal_handlers();
392
       while (true) {
   // poll background jobs
  int status;
393
394
395
396
          pid_t child_pid;
397
          while ((child_pid = s_waitpid(-1, &status, true)) > 0) {
398
             // Find which job child_pid belongs to
             for (size_t i = 0; i < vec_len(&job_list); i++) {
   job* job = vec_get(&job_list, i);
   bool in_this_job = false;
   for (size_t j = 0; j < job>num_pids; j++) {
      if (job>pids[j] == child_pid) {
399
400
401
402
403
404
                     in_this_job = true;
405
                     break;
406
407
408
409
                if (!in_this_job) {
410
                  continue;
411
412
413
                // If the process ended normally or via signal
414
                if (P_WIFEXITED(status) || P_WIFSIGNALED(status)) {
415
                   job->finished_count++;
```

```
416
               if (job->finished_count == job->num_pids) {
                 char buf[128];
417
                 saritation;
snprintf(buf, sizeof(buf), "Finished: ");
s_write(STDOUT_FILENO, buf, strlen(buf));
for (size_t cmdIdx = 0; cmdIdx < job->cmd->num_commands; cmdIdx++) {
418
419
420
421
                    char** argv = job->cmd->commands[cmdIdx];
                    int argIdx = 0;
422
423
                    while (argv[argIdx] != NULL) {
424
                     snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
                      s_write(STDOUT_FILENO, buf, strlen(buf));
425
                      argIdx++;
426
427
                   }
428
429
                 snprintf(buf, sizeof(buf), "\n");
430
                  s_write(STDOUT_FILENO, buf, strlen(buf));
431
                 vec_erase(&job_list, i);
432
            } else if (P_WIFSTOPPED(status) && job->state == RUNNING) {
433
               job->state = STOPPED;
434
               char buf[128];
435
               snprintf(buf, sizeof(buf), "Stopped: ");
s_write(STDOUT_FILENO, buf, strlen(buf));
436
437
               for (size_t cmdIdx = 0; cmdIdx < job->cmd->num_commands; cmdIdx++) {
438
439
                 char** argv = job->cmd->commands[cmdIdx];
                 int argIdx = 0;
440
                 while (argv[argIdx] != NULL) {
   snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
441
442
443
                    s_write(STDOUT_FILENO, buf, strlen(buf));
444
                    argIdx++;
445
                 }
446
               snprintf(buf, sizeof(buf), "\n");
448
               s_write(STDOUT_FILENO, buf, strlen(buf));
449
450
             break; // break from for-loop over job_list
451
          }
        }
452
453
454
        // prompt
455
        if (s_write(STDOUT_FILENO, PROMPT, strlen(PROMPT)) < 0) {</pre>
456
          u_perror("prompt s_write error");
457
          break:
458
459
        // parse user input
461
        char buffer[MAX_BUFFER_SIZE];
462
        ssize_t user_input = s_read(STDIN_FILENO, buffer, MAX_BUFFER_SIZE);
463
        if (user_input < 0) {</pre>
          u_perror("shell read error");
464
465
          break;
        } else if (user_input == 0) { // EOF case
466
467
         s_shutdown_pennos();
468
          break;
469
470
        buffer[user_input] = ' \setminus 0';
471
472
        if (buffer[user_input - 1] == '\n') {
473
          buffer[user_input - 1] = '\0';
474
475
476
        struct parsed command* cmd = NULL;
        int cmd_parse_res = parse_command(buffer, &cmd);
477
478
        if (cmd_parse_res != 0 || cmd == NULL) {
479
         P_ERRNO = P_EPARSE;
480
          u_perror("parse_command");
481
          continue;
482
483
484
        // handle the command
        if (cmd->num_commands == 0) {
485
486
         free(cmd);
487
           continue;
488
489
490
        child pid = execute command(cmd);
491
        if (child_pid < 0) {
492
          free (cmd);
        continue;
} else if (child_pid == 0) {
493
494
495
          free (cmd):
496
          continue;
497
498
499
         // If background, add the process to the job list.
500
        if (cmd->is_background) {
          // Create a new job entry.
job* new_job = malloc(sizeof(job));
501
502
```

```
503
           if (new_job == NULL) {
504
             perror("Error: mallocing new_job failed");
505
              free (cmd);
506
             continue;
507
           new_job->id = next_job_id++;
508
           new_job->pgid = child_pid; // For single commands, child's pid = pgid.
509
510
           new_job->num_pids = 1;
511
           new_job->pids = malloc(sizeof(pid_t));
           if (new_job->pids == NULL) {
  perror("Error: mallocing new_job->pids failed");
512
513
              free(new_job);
514
              free (cmd);
515
516
             continue;
517
           new_job->pids[0] = child_pid;
new_job->state = RUNNING;
new_job->cmd = cmd; // Retain command info; do not free here.
new_job->finished_count = 0;
518
519
520
521
522
           vec_push_back(&job_list, new_job);
523
524
           // Print job control information in the format: "[job_id] child_pid"
525
           char msg[128];
           snprintf(msg, sizeof(msg), "[%lu] %d\n", (unsigned long)new_job->id,
526
527
                      child_pid);
           if (s_write(STDOUT_FILENO, msg, strlen(msg)) == -1) {
  u_perror("s_write error");
528
529
530
531
         } else {
           // Foreground execution.
532
533
           current_fg_pid = child_pid;
534
           int status;
535
           s_waitpid(child_pid, &status, false);
536
537
           if (P_WIFSTOPPED(status)) {
              // Create a new job entry (this time for a stopped process)
job* new_job = malloc(sizeof(job));
538
539
              if (new_job == NULL) {
540
541
                perror("Error: mallocing new_job failed");
542
                free (cmd);
543
                continue;
544
              new_job->id = next_job_id++;
545
              new_job->pgid = child_pid; // For single commands, child's pid = pgid.
546
547
              new_job->num_pids = 1;
              new_job->pids = malloc(sizeof(pid_t));
548
549
              if (new_job->pids == NULL) {
                perror("Error: mallocing new_job->pids failed");
550
551
                free (new_job);
552
                free (cmd);
553
                continue;
554
             new_job->pids[0] = child_pid;
new_job->state = STOPPED;
new_job->cmd = cmd; // Retain command info; do not free here.
new_job->finished_count = 0;
555
556
557
558
              vec_push_back(&job_list, new_job);
559
560
561
              // Print stopped job
              char buf[128];
562
              snprintf(buf, sizeof(buf), "Stopped: ");
s_write(STDOUT_FILENO, buf, strlen(buf));
563
564
565
              for (size_t cmdIdx = 0; cmdIdx < new_job->cmd->num_commands; cmdIdx++) {
               char** argv = new_job->cmd->commands[cmdIdx];
                int argIdx = 0;
567
                while (argv[argIdx] != NULL) {
   snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
   s_write(STDOUT_FILENO, buf, strlen(buf));
568
569
570
571
                  argIdx++;
                }
573
574
              snprintf(buf, sizeof(buf), "\n");
575
              s_write(STDOUT_FILENO, buf, strlen(buf));
576
577
578
           current_fg_pid = 2;
579
580
            // Free cmd memory for foreground commands.
581
           // free(cmd); // TODO --> check if this is already freed, it may be
        }
582
583
584
585
      vec_destroy(&job_list);
586
      s_exit();
587
      return 0;
588 1
```

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 62 of file shell.c.

```
62
    // If there's a foreground process, forward SIGTSTP (stop) to it
64    if (current_fg_pid != 2) {
65         s_kill(current_fg_pid, 0); // P_SIGSTOP
66    }
67
68    if (s_write(STDOUT_FILENO, "\n", 1) == -1) {
69         u_perror("s_write error");
70    }
71 }
```

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 143 of file shell.c.

```
143
        // check for independently scheduled processes
144
       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) {
145
146
147
         148
149
       } else if (strcmp(cmd->commands[0][0], "busy") == 0) {
150
       return s_spawn(u_busy, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "echo") == 0) {
   return s_spawn(u_echo, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "ls") == 0) {
151
152
153
154
       return s_spawn(u_ls, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "touch") == 0) {
155
156
157
          return s_spawn(u_touch, cmd->commands[0], input_fd_script,
158
                             output_fd_script);
       } else if (strcmp(cmd->commands[0][0], "mv") == 0) {
159
       return s_spawn(u_mv, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "cp") == 0) {
160
161
       return s_spawn(u_cp, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "rm") == 0) {
162
163
       return s_spawn(u_rm, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "chmod") == 0) {
164
165
         return s_spawn(u_chmod, cmd->commands[0], input_fd_script,
166
167
                             output_fd_script);
       } else if (strcmp(cmd->commands[0][0], "ps") == 0) {
168
       return s_spawn(u_ps, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "kill") == 0) {
169
170
       return s_spawn(u_kill, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "zombify") == 0) {
171
172
173
         return s_spawn(u_zombify, cmd->commands[0], input_fd_script,
174
                            output_fd_script);
175
       } else if (strcmp(cmd->commands[0][0], "orphanify") == 0) {
176
          return s_spawn(u_orphanify, cmd->commands[0], input_fd_script,
177
                             output_fd_script);
       } else if (strcmp(cmd->commands[0][0], "hang") == 0) {
178
       return s_spawn(hang, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "nohang") == 0) {
179
180
181
          return s_spawn(nohang, cmd->commands[0], input_fd_script, output_fd_script);
       } else if (strcmp(cmd->commands[0][0], "recur") == 0) {
182
       return s_spawn(recur, cmd->commands[0], input_fd_script, output_fd_script);
} else if (strcmp(cmd->commands[0][0], "crash") == 0) {
183
184
185
         return s_spawn(crash, cmd->commands[0], input_fd_script, output_fd_script);
186
187
188
       // check for sub-routines
189
       if (strcmp(cmd->commands[0][0], "nice") == 0) {
190
         u_nice(cmd->commands[0]);
191
          return 0:
       } else if (strcmp(cmd->commands[0][0], "nice_pid") == 0) {
192
193
          u_nice_pid(cmd->commands[0]);
194
          return 0;
195
        } else if (strcmp(cmd->commands[0][0], "man") == 0) {
196
          u_man(cmd->commands[0]);
197
          return 0;
       } else if (strcmp(cmd->commands[0][0], "bg") == 0) {
198
199
         u_bg(cmd->commands[0]);
200
          return 0;
201
       } else if (strcmp(cmd->commands[0][0], "fg") == 0) {
          u_fg(cmd->commands[0]);
202
       return 0;
} else if (strcmp(cmd->commands[0][0], "jobs") == 0) {
203
204
205
         u_jobs(cmd->commands[0]);
          return 0;
207
       } else if (strcmp(cmd->commands[0][0], "logout") == 0) {
208
          u_logout(cmd->commands[0]);
209
          return 0;
210
       } else {
211
         return -1; // no matches, no scripts now
212
213
214
       return 0;
215 }
```

4.34.2.9 u_read_and_execute_script()

Helper function that reads a script file line by line, parses each line as a command, and executes it.

Parameters

arg standard (function name, NULL) args

Returns

NULL

```
Definition at line 224 of file shell.c.
225
       // read the script line by line, parse each line, and execute the command
226
      while (true) {
       char buffer[MAX_LINE_BUFFER_SIZE];
227
        fill_buffer_until_full_or_newline(script_fd, buffer);
if (buffer[0] == '\0') {
228
229
230
           break; // EOF case
231
232
233
        // parse the command
        struct parsed_command* cmd = NULL;
234
        int parse_result = parse_command(buffer, &cmd);
if (parse_result != 0 || cmd == NULL) {
235
236
         P_ERRNO = P_EPARSE;
u_perror("parse_command");
237
238
239
          free(cmd);
240
241
        // execute the command
242
243
        pid_t child_pid = u_execute_command(cmd);
244
        if (child_pid > 0) { // if process was spawned, wait for it to finish
245
         int status;
       s_waitpid(child_pid, &status, false);
} else if (child_pid < 0) { // nothing spawning so safe to free cmd</pre>
246
247
248
          free (cmd);
249
250
251
      s_exit(); // exit the script
252
253
      return NULL;
254 }
```

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 40 of file shell.c.

4.34.3.3 is_append

```
int is_append = 0
```

Definition at line 42 of file shell.c.

4.34.3.4 job_list

```
Vec job_list
```

Definition at line 35 of file shell.c.

4.34.3.5 next_job_id

```
jid_t next_job_id = 1
```

Definition at line 36 of file shell.c.

4.34.3.6 output_fd_script

```
int output_fd_script = -1
```

Definition at line 41 of file shell.c.

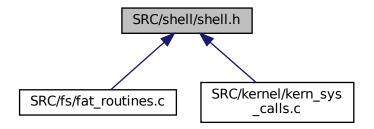
4.34.3.7 script_fd

```
int script_fd = -1
```

Definition at line 39 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 388 of file shell.c.

```
389
       job_list = vec_new(0, free_job_ptr);
390
391
       setup_terminal_signal_handlers();
392
393
       while (true) {
394
        // poll background jobs
395
         int status;
396
         pid_t child_pid;
         while ((child_pid = s_waitpid(-1, &status, true)) > 0) {
397
            // Find which job child_pid belongs to
for (size_t i = 0; i < vec_len(&job_list); i++) {
398
399
400
              job* job = vec_get(&job_list, i);
              bool in_this_job = false;
for (size_t j = 0; j < job->num_pids; j++) {
   if (job->pids[j] == child_pid) {
     in_this_job = true;
}
401
402
403
404
405
                   break;
406
407
408
              if (!in_this_job) {
409
410
                continue;
411
413
              \ensuremath{//} If the process ended normally or via signal
414
              if (P_WIFEXITED(status) || P_WIFSIGNALED(status)) {
415
                 job->finished_count++;
if (job->finished_count == job->num_pids) {
416
417
                   char buf[128];
418
                   snprintf(buf, sizeof(buf), "Finished: ");
419
                   s_write(STDOUT_FILENO, buf, strlen(buf));
420
                   for (size_t cmdIdx = 0; cmdIdx < job->cmd->num_commands; cmdIdx++) {
                     char** argv = job->cmd->commands[cmdIdx];
int argIdx = 0;
421
422
423
                     while (argv[argIdx] != NULL) {
```

```
snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
s_write(STDOUT_FILENO, buf, strlen(buf));
425
426
                        argIdx++;
                     }
427
428
                   snprintf(buf, sizeof(buf), "\n");
s_write(STDOUT_FILENO, buf, strlen(buf));
429
430
431
                   vec_erase(&job_list, i);
432
              } else if (P_WIFSTOPPED(status) && job->state == RUNNING) {
433
                 iob->state = STOPPED;
434
                 char buf[128];
435
                 snprintf(buf, sizeof(buf), "Stopped: ");
s_write(STDOUT_FILENO, buf, strlen(buf));
436
437
438
                 for (size_t cmdIdx = 0; cmdIdx < job->cmd->num_commands; cmdIdx++) {
                   char** argv = job->cmd->commands[cmdIdx];
int argIdx = 0;
439
440
                   while (argv[argIdx] != NULL) {
    snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
441
442
                     s_write(STDOUT_FILENO, buf, strlen(buf));
443
444
                     argIdx++;
445
                   }
446
                snprintf(buf, sizeof(buf), "\n");
s_write(STDOUT_FILENO, buf, strlen(buf));
447
448
450
              break; // break from for-loop over job_list
451
452
         }
453
454
         // prompt
455
         if (s_write(STDOUT_FILENO, PROMPT, strlen(PROMPT)) < 0) {</pre>
456
          u_perror("prompt s_write error");
457
458
459
         // parse user input
460
         char buffer[MAX_BUFFER_SIZE];
461
462
         ssize_t user_input = s_read(STDIN_FILENO, buffer, MAX_BUFFER_SIZE);
463
         if (user_input < 0) {</pre>
464
           u_perror("shell read error");
465
           break;
         } else if (user_input == 0) { // EOF case
466
467
           s_shutdown_pennos();
468
           break;
469
470
         buffer[user_input] = '\0';
if (buffer[user_input - 1] == '\n') {
  buffer[user_input - 1] = '\0';
471
472
473
474
475
476
         struct parsed_command* cmd = NULL;
         int cmd_parse_command(buffer, &cmd);
if (cmd_parse_res != 0 || cmd == NULL) {
    P_ERRNO = P_EPARSE;
    u_perror("parse_command");
477
478
479
480
481
           continue;
482
483
         // handle the command
484
485
         if (cmd->num_commands == 0) {
486
           free(cmd);
487
           continue;
488
489
490
         child_pid = execute_command(cmd);
         if (child_pid < 0) {
491
492
           free(cmd);
493
            continue;
494
         } else if (child_pid == 0) {
495
           free(cmd);
496
           continue;
497
498
499
         // If background, add the process to the job list.
500
         if (cmd->is_background) {
501
            // Create a new job entry.
            job* new_job = malloc(sizeof(job));
if (new_job == NULL) {
502
503
             perror("Error: mallocing new_job failed");
504
505
              free(cmd);
506
              continue;
507
508
            new_job->id = next_job_id++;
           new_job->pgid = child_pid; // For single commands, child's pid = pgid.
509
510
           new_job->num_pids = 1;
```

```
511
           new_job->pids = malloc(sizeof(pid_t));
           if (new_job->pids == NULL) {
512
513
             perror("Error: mallocing new_job->pids failed");
514
             free(new_job);
515
             free (cmd):
516
             continue:
517
518
           new_job->pids[0] = child_pid;
           new_job->state = RUNNING;
new_job->cmd = cmd; // Retain command info; do not free here.
519
520
           new_job->finished_count = 0;
521
           vec_push_back(&job_list, new_job);
522
523
524
           // Print job control information in the format: "[job_id] child_pid"
525
           char msg[128];
526
           snprintf(msg, \ sizeof(msg), \ "[\$lu] \ \$d\n", \ (unsigned \ long) \ new\_job->id,
           child_pid);
if (s_write(STDOUT_FILENO, msg, strlen(msg)) == -1) {
  u_perror("s_write error");
527
528
529
530
531
         } else {
532
           // Foreground execution.
533
           current_fg_pid = child_pid;
534
           int status:
535
           s_waitpid(child_pid, &status, false);
536
537
           if (P_WIFSTOPPED(status)) {
538
             // Create a new job entry (this time for a stopped process)
             job* new_job = malloc(sizeof(job));
539
             if (new_job == NULL) {
540
                perror("Error: mallocing new_job failed");
541
542
                free (cmd);
543
               continue;
544
             new_job->id = next_job_id++;
new_job->pgid = child_pid; // For single commands, child's pid = pgid.
545
546
             new_job->num_pids = 1;
547
             new_job->pids = malloc(sizeof(pid_t));
548
549
             if (new_job->pids == NULL) {
550
              perror("Error: mallocing new_job->pids failed");
551
                free(new_job);
552
               free (cmd);
553
               continue;
554
555
             new_job->pids[0] = child_pid;
556
             new_job->state = STOPPED;
557
             new_job->cmd = cmd; // Retain command info; do not free here.
558
             new_job->finished_count = 0;
559
             vec_push_back(&job_list, new_job);
560
561
             // Print stopped job
562
             char buf[128];
             snprintf(buf, sizeof(buf), "Stopped: ");
s_write(STDOUT_FILENO, buf, strlen(buf));
for (size_t cmdIdx = 0; cmdIdx < new_job->cmd->num_commands; cmdIdx++) {
563
564
565
               char** argv = new_job->cmd->commands[cmdIdx];
566
                int argIdx = 0;
567
               while (argv[argIdx] != NULL) {
   snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
568
569
570
                  s_write(STDOUT_FILENO, buf, strlen(buf));
571
                  argIdx++;
572
               }
573
574
             snprintf(buf, sizeof(buf), "\n");
575
             s_write(STDOUT_FILENO, buf, strlen(buf));
576
577
578
           current_fq_pid = 2;
579
580
           // Free cmd memory for foreground commands.
581
           // free(cmd); // TODO --> check if this is already freed, it may be
582
      }
583
584
585
      vec_destroy(&job_list);
      s_exit();
587
      return 0;
588 }
```

4.35.2 Variable Documentation

4.35.2.1 is_append

```
int is_append [extern]
```

Definition at line 42 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_pcb.h"
#include "../kernel/kern_sys_calls.h"
#include "../kernel/scheduler.h"
#include "../kernel/signal.h"
#include "../lib/Vec.h"
#include "../lib/spthread.h"
#include "builtins.h"
#include "lib/pennos-errno.h"
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "Job.h"
Include dependency graph for shell built ins.c:
```

Americand John

| Solidation of the continue o

Functions

```
void * u_cat (void *arg)
```

The standard 'cat' program built-in.

void * u_sleep (void *arg)

The standard 'sleep' program built-in.

void * u_busy (void *arg)

Built-in that hangs indefinitely.

void * u_echo (void *arg)

Standard 'echo' program built-in that reads a string and echos it backs.

void * u ls (void *arg)

Standard 'Is' program built-in that lists files in working directory.

void * u_chmod (void *arg)

Standard 'chmod' progrom built-in that changes the permissions of a given file.

void * u_touch (void *arg)

Standard 'touch' program built-in that creates empty files or updates timestamps.

void * u mv (void *arg)

Standard 'mv' program built-in that renames files.

void * u_cp (void *arg)

Standard 'cp' program built-in that copies files.

void * u rm (void *arg)

Standard 'rm' program built-in that removes files.

void * u_ps (void *arg)

Standard 'ps' program built-in that lists processes in PennOS.

void * u_kill (void *arg)

Standard 'kill' program built-in that sends the specified signal to a process.

void * u_nice (void *arg)

Spawns a new process for the given command and sets it priority to the given priority.

void * u_nice_pid (void *arg)

Adjusts priority level of an existing process.

void * u man (void *arg)

Lists all available commands in PennOS in terminal.

job * findJobByIdOrCurrent (const char *arg)

Helper function. Finds a job by its id or the current job.

void * u_bg (void *arg)

Resumes the most recently stopped background jobs or a specified one.

void * u_fg (void *arg)

Brings the most recently stopped or background job to the foreground or a specified one.

void * u_jobs (void *arg)

Lists all jobs.

void * u_logout (void *arg)

Exits the shell and shuts down PennOS.

void * zombie_child (void *arg)

Helper for zombify.

void * u_zombify (void *arg)

Built-in that tests zombifying functionality of the kernel.

void * orphan_child (void *arg)

Helper for orphanify.

void * u_orphanify (void *arg)

Built-in that tests orphanifying functionality of the kernel.

Variables

- Vec job list
- · pid t current fg pid
- 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 findJobByldOrCurrent()

Helper function. Finds a job by its id or the current job.

Example Usage: findJobByIdOrCurrent(a string representing the job id)

Definition at line 341 of file shell built ins.c.

```
342
       return NULL;
}
        if (vec_len(&job_list) == 0) {
343
344
345
        if (arg != NULL) {
346
        // parse numeric
char* endPtr = NULL;
348
          long val = strtol(arg, &endPtr, 10);
if (*endPtr != '\0' || val < 1) {</pre>
349
350
          return NULL;
351
352
353
          for (size_t i = 0; i < vec_len(&job_list); i++) {</pre>
          job* job_ptr = (job*)vec_get(&job_list, i);
if ((jid_t)val == job_ptr->id) {
354
355
356
                return job_ptr;
357
358
359
          return NULL;
360
361
        // Look for most recently stopped job first
for (size_t i = vec_len(&job_list); i > 0; i--) {
   job* job_ptr = (job*)vec_get(&job_list, i - 1);
   if (job_ptr->state == STOPPED) {
362
363
364
365
366
             return job_ptr;
367
368
369
       return (job*)vec_get(&job_list, vec_len(&job_list) - 1);
370
```

4.36.1.2 orphan_child()

Helper for orphanify.

Definition at line 581 of file shell_built_ins.c.

```
581
582 while (1)
583 ;
584 s_exit();
585 }
```

4.36.1.3 u_bg()

Resumes the most recently stopped background jobs or a specified one.

Resumes the most recently stopped job in the background, or the job specified by job id.

Definition at line 377 of file shell_built_ins.c.

```
378
      char buf[128];
379
      char** argv = (char**)arg;
      const char* jobArg = argv[1]; // NULL if no ID was given
job* job_ptr = findJobByIdOrCurrent(jobArg);
380
381
382
      if (!job_ptr) {
383
        snprintf(buf, sizeof(buf), "bg: no such job\n");
384
        s_write(STDERR_FILENO, buf, strlen(buf));
385
        return NULL;
386
387
      if (job_ptr->state == STOPPED) {
388
        job_ptr->state = RUNNING;
389
        snprintf(buf, sizeof(buf), "Running: ");
        s_write(STDOUT_FILENO, buf, strlen(buf));
for (size_t cmdIdx = 0; cmdIdx < job_ptr->cmd->num_commands; cmdIdx++) {
390
391
         char** argv = job_ptr->cmd->commands[cmdIdx];
int argIdx = 0;
392
393
394
          while (argv[argIdx] != NULL) {
            snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
s_write(STDOUT_FILENO, buf, strlen(buf));
395
396
397
            argIdx++;
          }
398
399
400
        snprintf(buf, sizeof(buf), "\n");
401
        s_write(STDOUT_FILENO, buf, strlen(buf));
402
        // P SIGCONT is 1
403
        s_kill(job_ptr->pids[0], P_SIGCONT);
404
        return NULL;
     } else if (job_ptr->state == RUNNING) {
405
406
       snprintf(buf, sizeof(buf), "bg: job [%lu] is already running\n",
407
                  (unsigned long) job_ptr->id);
408
        s_write(STDOUT_FILENO, buf, strlen(buf));
409
        return NULL;
410
     } else {
       411
412
        s_write(STDOUT_FILENO, buf, strlen(buf));
413
414
        return NULL;
     }
415
416 }
```

4.36.1.4 u_busy()

```
void* u_busy (
     void * arg )
```

Built-in that hangs indefinitely.

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

Definition at line 73 of file shell built ins.c.

```
73
74 while (1)
75 ;
76 s_exit();
77 return NULL;
78 }
```

4.36.1.5 u_cat()

The standard 'cat' program built-in.

The usual cat program.

Definition at line 42 of file shell built ins.c.

```
42 {
43 cat(arg);
44 s_exit();
45 return NULL;
46 }
```

4.36.1.6 u_chmod()

Standard 'chmod' progrom built-in that changes the permissions of a given file.

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.

Definition at line 104 of file shell_built_ins.c.

```
104 {
105 chmod(arg);
106 s_exit();
107 return NULL;
108 }
```

4.36.1.7 u_cp()

Standard 'cp' program built-in that copies files.

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
- ${\tt src_file}$ 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 132 of file shell_built_ins.c.

```
132 {
133 cp(arg);
134 s_exit();
135 return NULL;
136 }
```

4.36.1.8 u_echo()

Standard 'echo' program built-in that reads a string and echos it backs.

Echo back an input string.

Definition at line 84 of file shell built ins.c.

```
84 {
85     s_echo(arg);
86     s_exit();
87     return NULL;
88 }
```

4.36.1.9 u fg()

Brings the most recently stopped or background job to the foreground or a specified one.

Brings the most recently stopped or background job to the foreground, or the job specified by job_id.

Definition at line 422 of file shell_built_ins.c.

```
char buf[128];
      char** argv = (char**)arg;

const char* jobArg = argv[1]; // NULL if no ID was given
424
425
      job* job_ptr = findJobByIdOrCurrent(jobArg);
if (!job_ptr) {
426
427
428
        snprintf(buf, sizeof(buf), "fg: no such job\n");
429
        s_write(STDERR_FILENO, buf, strlen(buf));
430
         return NULL;
431
432
433
      if (job ptr->state == FINISHED) {
       snprintf(buf, sizeof(buf), "fg: job [%lu] is already finished\n",
434
435
                    (unsigned long) job_ptr->id);
        s_write(STDOUT_FILENO, buf, strlen(buf));
436
437
         return NULL;
438
439
440
      if (job_ptr->state == STOPPED) {
441
        job_ptr->state = RUNNING;
442
         snprintf(buf, sizeof(buf), "Restarting: ");
         s_write(STDOUT_FILENO, buf, strlen(buf));
443
         for (size_t cmdIdx = 0; cmdIdx < job_ptr->cmd->num_commands; cmdIdx++) {
   char** argv = job_ptr->cmd->commands[cmdIdx];
   int argIdx = 0;
444
445
446
           while (argv[argIdx] != NULL) {
    snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
448
449
              s_write(STDOUT_FILENO, buf, strlen(buf));
450
              argIdx++;
           }
451
452
453
         snprintf(buf, sizeof(buf), "\n");
454
         s_write(STDOUT_FILENO, buf, strlen(buf));
455
         // P_SIGCONT is 1
456
         s_kill(job_ptr->pids[0], P_SIGCONT);
457
      } else {
         snprintf(buf, sizeof(buf), "Bringing to foreground: ");
458
         swrite(STDOUT_FILENO, buf, strlen(buf));
for (size_t cmdIdx = 0; cmdIdx < job_ptr->cmd->num_commands; cmdIdx++) {
459
460
461
           char** argv = job_ptr->cmd->commands[cmdIdx];
462
           int argIdx = 0;
           while (argv[argIdx] != NULL) {
  snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
  s_write(STDOUT_FILENO, buf, strlen(buf));
463
464
465
466
              argIdx++;
```

```
467
468
469
         snprintf(buf, sizeof(buf), "\n");
470
471
472
      current_fg_pid = job_ptr->pids[0];
473
474
      while (true) {
        int status = 0;
pid_t wpid = s_waitpid(job_ptr->pgid, &status, false);
475
476
         if (wpid < 0) {
477
478
          if (P_ERRNO == P_EINTR) {
479
            continue;
480
481
           break;
482
         if (P_WIFEXITED(status) || P_WIFSIGNALED(status)) {
483
           job_ptr->state = FINISHED;
484
           // Remove finished job from list
485
486
           for (size_t i = 0; i < vec_len(&job_list); i++) {</pre>
487
             if ((job*)vec_get(&job_list, i) == job_ptr) {
488
               vec_erase(&job_list, i);
489
               break;
490
             }
491
492
           break;
493
494
         if (P_WIFSTOPPED(status)) {
           job_ptr->state = STOPPED;
snprintf(buf, sizeof(buf), "Stopped: ");
s_write(STDOUT_FILENO, buf, strlen(buf));
495
496
497
498
           print_parsed_command(job_ptr->cmd);
499
500
      }
501
502
503  // back to shell
504  current_fg_pid = 2;
505
      return NULL;
506 }
```

4.36.1.10 u_jobs()

Lists all jobs.

Example Usage: jobs

Definition at line 511 of file shell_built_ins.c.

```
512
      char buf[128];
513
      if (vec_is_empty(&job_list)) {
  return NULL;
514
515
516
517
      for (size_t idx = 0; idx < vec_len(&job_list); idx++) {</pre>
       job* job_ptr = (job*)vec_get(&job_list, idx);
518
519
        const char* state = "unknown";
520
521
        if (job_ptr->state == RUNNING) {
          state = "running";
522
523
        } else if (job_ptr->state == STOPPED) {
        state = "stopped";
} else if (job_ptr->state == FINISHED) {
   state = "finished";
524
525
526
527
528
529
        snprintf(buf, sizeof(buf), "[%lu] ", (unsigned long)job_ptr->id);
530
         s_write(STDOUT_FILENO, buf, strlen(buf));
        for (size_t cmdIdx = 0; cmdIdx < job_ptr->cmd->num_commands; cmdIdx++) {
531
         char** argv = job_ptr->cmd->commands[cmdIdx];
int argIdx = 0;
532
533
534
          while (argv[argIdx] != NULL) {
             snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
```

4.36.1.11 u kill()

```
void* u_kill (
     void * arg )
```

Standard 'kill' program built-in that sends the specified signal to a process.

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.

Definition at line 161 of file shell_built_ins.c.

```
161
162
       char** argv = (char**)arg;
       163
164
165
       char err_buf[128];
166
       // Check if the first argument specifies a signal
if (argv[start_index] && argv[start_index][0] == '-') {
167
168
169
        if (strcmp(argv[start_index], "-term") == 0) {
170
           sig = 2;
         } else if (strcmp(argv[start_index], "-stop") == 0) {
171
           sig = 0;
172
         } else if (strcmp(argv[start_index], "-cont") == 0) {
  sig = 1;
173
174
         } else {
  // Construct error message
175
176
177
            s_exit();
178
            return NULL;
179
180
         start_index++;
182
183
       // Process each PID argument using strtol
184
       for (int i = start_index; argv[i] != NULL; i++) {
185
         char* endptr:
186
          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>
187
188
            if (s_write(STDERR_FILENO, err_buf, strlen(err_buf)) == -1) {
    u_perror("s_write error");
189
190
191
            }
192
            continue;
193
194
         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>
195
196
197
198
199
200
         }
201
       }
202
       s_exit();
203
      return NULL;
204 }
```

4.36.1.12 u_logout()

Exits the shell and shuts down PennOS.

Exits the shell and shutsdown PennOS.

Definition at line 549 of file shell_built_ins.c.

```
549 {
550 s_shutdown_pennos();
551 return NULL;
552 }
```

4.36.1.13 u_ls()

Standard 'Is' program built-in that lists files in working directory.

Lists all files in the working directory. For extra credit, it should support relative and absolute file paths.

Definition at line 94 of file shell_built_ins.c.

```
94 {
95    ls(arg);
96    s_exit();
97    return NULL;
98 }
```

4.36.1.14 u_man()

Lists all available commands in PennOS in terminal.

Lists all available commands.

Definition at line 300 of file shell_built_ins.c.

```
301
302
303
304
305
306
              "ls : lists all files in the working directory\n"
"touch f1 f2 ... : for each file, creates empty file if it doesn't "
307
308
              "exist yet, otherwise updates its timestamp\n"
"mv f1 f2 : renames f1 to f2 (overwrites f2 if it exists)\n"
"cp f1 f2 : copies f1 to f2 (overwrites f2 if it exists)\n"
309
310
311
              "rm f1 f2 ...
"chmod +_ f1
                                      : removes the input list of files\n"
: changes fl permissions to +_ specifications "
312
313
              "(+x, +rw, etc)\n"
314
              "ps : lists all processes on PennOS, displaying PID, " "PPID, priority, status, and command name\n" "kill (-__) pidl pid 2 : sends specified signal (term default) to list "
315
316
317
318
              "of processes\n"
```

```
319
          "nice n command
                                  : spawns a new process for command and sets its "
320
          "priority to n \ "
321
          "nice_pid n pid
                                  : adjusts the priority level of an existing "
          "process to n \ n"
322
323
          "man
                                 : lists all available commands in PennOS\n"
          "bg
324
                                  : resumes most recently stopped process in "
          "background or the one specified by job_id\n"
325
326
          "fg
                                  : brings most recently stopped or background job "
327
          "to foreground or the one specifed by job_id\n"
                             : lists all jobs\n"
: exits the shell and shuts down PennOS\n"
328
          "jobs
          "logout
329
          "zombify
                                 : creates a child process that becomes a zombie\n"
330
                                 : creates a child process that becomes an '
331
          "orphanify
332
          "orphan\n";
333
334
     s_write(STDOUT_FILENO, man_string, strlen(man_string));
335
     return NULL:
336 }
```

4.36.1.15 u mv()

Standard 'mv' program built-in that renames files.

Rename a file. If the dst_file file already exists, overwrite it.

Definition at line 123 of file shell_built_ins.c.

```
123 {
124 mv(arg);
125 s_exit();
126 return NULL;
127 }
```

4.36.1.16 u_nice()

Spawns a new process for the given command and sets it priority to the given priority.

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

Definition at line 255 of file shell_built_ins.c.

```
255
      char* endptr;
257
      int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10);
if (*endptr != '\0' || errno != 0 || new_priority > 2 ||
258
259
          new_priority < 0) { // error catch</pre>
2.60
261
        return NULL;
262
263
264
      char* command = ((char**)arg)[2];
265
      void* (*ufunc)(void*) = get_associated_ufunc(command);
      if (ufunc == NULL) {
  return NULL; // no matches, don't spawn
266
267
268
269
270
      pid_t new_proc_pid = s_spawn(ufunc, &((char**)arg)[2], 0, 1);
271
272
      if (new_proc_pid != -1) { // non-error case
273
        s_nice(new_proc_pid, new_priority);
274
275
276
      return NULL;
```

4.36.1.17 u_nice_pid()

Adjusts priority level of an existing process.

Adjust the priority level of an existing process.

Definition at line 282 of file shell_built_ins.c.

```
282
283
         char* endptr:
284
         errno = 0;
        int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10);
if (*endptr != '\0' || errno != 0) { // error catch
285
286
          return NULL;
287
288
        pid_t pid = (pid_t)strtol(((char**)arg)[2], &endptr, 10);
if (*endptr != '\0' || errno != 0) {
    return NULL;
289
290
291
292
293 s_nice(pid, new_priority);
294 return NULL;
295 }
```

4.36.1.18 u_orphanify()

Built-in that tests orphanifying functionality of the kernel.

Used to test orphanifying functionality of your kernel.

Definition at line 591 of file shell_built_ins.c.

```
591 {
592 char* orphan_child_argv[] = {"orphan_child", NULL};
593 s_spawn(orphan_child, orphan_child_argv, STDIN_FILENO, STDOUT_FILENO);
594 s_exit();
595 return NULL;
596 }
```

4.36.1.19 u_ps()

Standard 'ps' program built-in that lists processes in PennOS.

List all processes on PennOS, displaying PID, PPID, priority, status, and command name.

Definition at line 151 of file shell_built_ins.c.

```
151
152 s_ps(arg);
153 s_exit();
154 return NULL;
155 }
```

4.36.1.20 u_rm()

Standard 'rm' program built-in that removes files.

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.)

Definition at line 141 of file shell_built_ins.c.

```
141 {
142 rm(arg);
143 s_exit();
144 return NULL;
145 }
```

4.36.1.21 u_sleep()

The standard 'sleep' program built-in.

Sleep for n seconds.

Definition at line 51 of file shell_built_ins.c.

```
51
52
      char* endptr;
      errno = 0;
53
      if (((char**)arg)[1] == NULL) { // no arg case
55
         return NULL;
56
57
     int sleep_secs = (int)strtol(((char**)arg)[1], &endptr, 10);
if (*endptr != '\0' || errno != 0 || sleep_secs <= 0) {
    s_exit();</pre>
58
         return NULL;
62 }
6.3
int sleep_ticks = sleep_secs * 10;
s_sleep(sleep_ticks);
    s_exit();
return NULL;
68 }
```

4.36.1.22 u_touch()

```
void* u_touch (
     void * arg )
```

Standard 'touch' program built-in that creates empty files or updates timestamps.

For each file, create an empty file if it doesn't exist, else update its timestamp.

Definition at line 114 of file shell_built_ins.c.

```
114
115     touch(arg);
116     s_exit();
117     return NULL;
118 }
```

4.36.1.23 u_zombify()

```
void* u_zombify (
     void * arg )
```

Built-in that tests zombifying functionality of the kernel.

Used to test zombifying functionality of your kernel.

Definition at line 570 of file shell_built_ins.c.

```
570 {
571 char* zombie_child_argv[] = {"zombie_child", NULL};
572 s_spawn(zombie_child, zombie_child_argv, STDIN_FILENO, STDOUT_FILENO);
573 while (1)
574 ;
575 return NULL;
576 }
```

4.36.1.24 zombie_child()

Helper for zombify.

Definition at line 561 of file shell_built_ins.c.

```
561 {
562 s_exit();
563 return NULL;
564 }
```

4.36.2 Variable Documentation

4.36.2.1 current_fg_pid

```
pid_t current_fg_pid [extern]
```

Definition at line 31 of file kern_sys_calls.c.

4.36.2.2 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

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

Definition at line 221 of file shell_built_ins.c.

```
if (strcmp(func, "cat") == 0) {
222
     return u_cat;
} else if (strcmp(func, "sleep") == 0) {
223
224
225
       return u sleep;
     } else if (strcmp(func, "busy") == 0) {
226
       return u_busy;
228 } else if (strcmp(func, "echo") == 0) {
229    return u_echo;
230    } else if (strcmp(func, "ls") == 0) {
231
       return u_ls;
     } else if (strcmp(func, "touch") == 0) {
233
       return u_touch;
234 } else if (strcmp(func, "mv") == 0) {
237
       return u cp;
238 } else if (strcmp(func, "rm") == 0) {
       return u_rm;
240 } else if (strcmp(func, "chmod") == 0) {
241
       return u_chmod;
242 } else if (strcmp(func, "ps") == 0) {
243
       return u_ps;
244
     } else if (strcmp(func, "kill") == 0) {
245
       return u_kill;
246
247
248
     return NULL; // no matches case
249 }
```

4.36.2.3 job_list

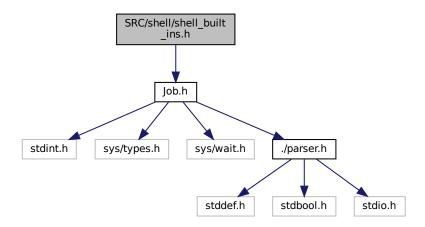
```
Vec job_list [extern]
```

Definition at line 35 of file shell.c.

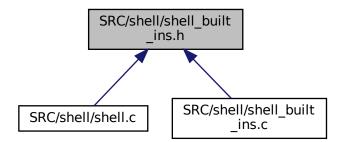
4.37 SRC/shell/shell built ins.h File Reference

```
#include "Job.h"
```

Include dependency graph for shell_built_ins.h:



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



Functions

void * u_cat (void *arg)

The usual cat program.

void * u_sleep (void *arg)

Sleep for n seconds.

void * u_busy (void *arg)

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

void * u_echo (void *arg)

Echo back an input string.

void * u_ls (void *arg)

Lists all files in the working directory. For extra credit, it should support relative and absolute file paths.

void * u_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)

Standard 'cp' program built-in that copies files.

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.

job * findJobByIdOrCurrent (const char *arg)

Helper function. Finds a job by its id or the current job.

void * u_bg (void *arg)

Resumes the most recently stopped job in the background, or the job specified by job_id.

void * u_fg (void *arg)

Brings the most recently stopped or background job to the foreground, or the job specified by job_id.

void * u_jobs (void *arg)

Lists all jobs.

void * u_logout (void *arg)

Exits the shell and shutsdown PennOS.

void * u_zombify (void *arg)

Used to test zombifying functionality of your kernel.

void * u_orphanify (void *arg)

Used to test orphanifying functionality of your kernel.

4.37.1 Function Documentation

4.37.1.1 findJobByldOrCurrent()

Helper function. Finds a job by its id or the current job.

Example Usage: findJobByIdOrCurrent(a string representing the job id)

Definition at line 341 of file shell_built_ins.c.

```
342
       if (vec_len(&job_list) == 0) {
343
        return NULL;
344
345
       if (arg != NULL) {
346
347
         // parse numeric
348
         char* endPtr = NULL;
349
         long val = strtol(arg, &endPtr, 10);
         if (*endPtr != '\0' || val < 1) {
350
351
            return NULL:
352
353
         for (size_t i = 0; i < vec_len(&job_list); i++) {</pre>
            job* job_ptr = (job*) vec_get(&job_list, i);
if ((jid_t) val == job_ptr->id) {
354
355
356
              return job_ptr;
357
            }
358
359
         return NULL;
360
361
       // Look for most recently stopped job first
for (size_t i = vec_len(&job_list); i > 0; i--) {
   job* job_ptr = (job*)vec_get(&job_list, i - 1);
362
363
364
         if (job_ptr->state == STOPPED) {
365
366
            return job_ptr;
367
368
369
370
       return (job*)vec_get(&job_list, vec_len(&job_list) - 1);
```

4.37.1.2 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)

Resumes the most recently stopped job in the background, or the job specified by job_id.

Definition at line 377 of file shell_built_ins.c.

```
378
      char buf[128];
379
      char** argv = (char**)arg;
380
      const char* jobArg = argv[1]; // NULL if no ID was given
      job* job_ptr = findJobByIdOrCurrent(jobArg);
382
      if (!job_ptr) {
383
        snprintf(buf, sizeof(buf), "bg: no such job\n");
384
        s_write(STDERR_FILENO, buf, strlen(buf));
        return NULL;
385
386
387
      if (job_ptr->state == STOPPED) {
388
         job_ptr->state = RUNNING;
389
         snprintf(buf, sizeof(buf), "Running: ");
        s_write(STDOUT_FILENO, buf, strlen(buf));
for (size_t cmdIdx = 0; cmdIdx < job_ptr->cmd->num_commands; cmdIdx++) {
390
391
          char** argv = job_ptr->cmd->commands[cmdIdx];
392
           int argIdx = 0;
393
394
           while (argv[argIdx] != NULL) {
            snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
s_write(STDOUT_FILENO, buf, strlen(buf));
395
396
397
             argIdx++:
398
399
400
        snprintf(buf, sizeof(buf), "\n");
401
        s_write(STDOUT_FILENO, buf, strlen(buf));
402
        // P_SIGCONT is 1
403
        s_kill(job_ptr->pids[0], P_SIGCONT);
404
        return NULL:
      } else if (job_ptr->state == RUNNING) {
405
        snprintf(buf, sizeof(buf), "bg: job [%lu] is already running\n",
```

```
407
                 (unsigned long) job_ptr->id);
408
       s_write(STDOUT_FILENO, buf, strlen(buf));
409
       return NULL;
     } else {
410
       snprintf(buf, sizeof(buf), "bg: job [%lu] not stopped\n",
411
412
                 (unsigned long) job_ptr->id);
       s_write(STDOUT_FILENO, buf, strlen(buf));
413
414
       return NULL;
415 }
416 }
```

4.37.1.3 u_busy()

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

Example Usage: busy

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

Definition at line 73 of file shell_built_ins.c.

```
73 {
74 while (1)
75 ;
76 s_exit();
77 return NULL;
78 }
```

4.37.1.4 u_cat()

The usual 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)

The usual cat program.

Definition at line 42 of file shell_built_ins.c.

```
42 {
43 cat(arg);
44 s_exit();
45 return NULL;
46 }
```

4.37.1.5 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)

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.

Definition at line 104 of file shell_built_ins.c.

```
104 {
105 chmod(arg);
106 s_exit();
107 return NULL;
108 }
```

4.37.1.6 u_cp()

Standard 'cp' program built-in that copies files.

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 132 of file shell built ins.c.

```
132 {
133 cp(arg);
134 s_exit();
135 return NULL;
136 }
```

4.37.1.7 u_echo()

Echo back an input string.

Example Usage: echo Hello World

Echo back an input string.

Definition at line 84 of file shell built ins.c.

```
84 {
85     s_echo(arg);
86     s_exit();
87     return NULL;
88 }
```

4.37.1.8 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)

Brings the most recently stopped or background job to the foreground, or the job specified by job_id.

Definition at line 422 of file shell built ins.c.

```
423
     char buf[128];
     char** argv = (char**)arg;
424
     const char* jobArg = argv[1]; // NULL if no ID was given
425
     job* job_ptr = findJobByIdOrCurrent(jobArg);
426
     if (!job_ptr) {
427
      snprintf(buf, sizeof(buf), "fg: no such job\n");
428
429
       s_write(STDERR_FILENO, buf, strlen(buf));
430
       return NULL;
431
432
     if (job_ptr->state == FINISHED) {
433
      434
435
436
       s_write(STDOUT_FILENO, buf, strlen(buf));
437
       return NULL;
438
439
440
     if (job_ptr->state == STOPPED) {
       job_ptr->state = RUNNING;
441
442
       snprintf(buf, sizeof(buf), "Restarting: ");
443
        s_write(STDOUT_FILENO, buf, strlen(buf));
444
       for (size_t cmdIdx = 0; cmdIdx < job_ptr->cmd->num_commands; cmdIdx++) {
445
        char** argv = job_ptr->cmd->commands[cmdIdx];
int argIdx = 0;
446
447
         while (argv[argIdx] != NULL)
          snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
s_write(STDOUT_FILENO, buf, strlen(buf));
448
449
450
           argIdx++;
         }
451
452
453
       snprintf(buf, sizeof(buf), "\n");
454
       s_write(STDOUT_FILENO, buf, strlen(buf));
455
       // P_SIGCONT is 1
456
       s_kill(job_ptr->pids[0], P_SIGCONT);
457
     } else {
458
       snprintf(buf, sizeof(buf), "Bringing to foreground: ");
459
       s_write(STDOUT_FILENO, buf, strlen(buf));
       for (size_t cmdIdx = 0; cmdIdx < job_ptr->cmd->num_commands; cmdIdx++) {
```

```
461
            char** argv = job_ptr->cmd->commands[cmdIdx];
            int argIdx = 0;
            while (argv[argIdx] != NULL) {
   snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
   s_write(STDOUT_FILENO, buf, strlen(buf));
463
464
465
466
              argIdx++;
467
468
469
          snprintf(buf, sizeof(buf), "\n");
470
471
472
       current_fg_pid = job_ptr->pids[0];
473
474
475
          int status = 0;
          pid_t wpid = s_waitpid(job_ptr->pgid, &status, false);
476
          if (wpid < 0) {
  if (P_ERRNO == P_EINTR) {</pre>
477
478
479
              continue;
480
481
            break;
482
          if (P_WIFEXITED(status) || P_WIFSIGNALED(status)) {
   job_ptr->state = FINISHED;
483
484
            // Remove finished job from list
for (size_t i = 0; i < vec_len(&job_list); i++) {</pre>
485
486
487
              if ((job*)vec_get(&job_list, i) == job_ptr) {
488
                vec_erase(&job_list, i);
489
                 break;
              }
490
491
492
            break;
493
          if (P_WIFSTOPPED(status)) {
494
            job_ptr->state = STOPPED;
snprintf(buf, sizeof(buf), "Stopped: ");
s_write(STDOUT_FILENO, buf, strlen(buf));
495
496
497
498
            print_parsed_command(job_ptr->cmd);
499
            break;
500
501
       }
502
       // back to shell
503
      current_fg_pid = 2;
504
505
       return NULL;
506 }
```

4.37.1.9 u_jobs()

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

Lists all jobs.

Example Usage: jobs

Definition at line 511 of file shell_built_ins.c.

```
511
512
       char buf[128];
       if (vec_is_empty(&job_list)) {
  return NULL;
513
514
515
516
       for (size_t idx = 0; idx < vec_len(&job_list); idx++) {</pre>
518
        job* job_ptr = (job*)vec_get(&job_list, idx);
519
         const char* state = "unknown";
if (job_ptr->state == RUNNING) {
520
521
         state = "running";
} else if (job_ptr->state == STOPPED) {
522
523
         state = "stopped";
} else if (job_ptr->state == FINISHED) {
   state = "finished";
524
525
526
527
528
          snprintf(buf, sizeof(buf), "[%lu] ", (unsigned long)job_ptr->id);
```

```
530
         s_write(STDOUT_FILENO, buf, strlen(buf));
531
         for (size_t cmdIdx = 0; cmdIdx < job_ptr->cmd->num_commands; cmdIdx++) {
532
           char** argv = job_ptr->cmd->commands[cmdIdx];
           int argIdx = 0;
533
           while (argv[argIdx] != NULL) {
   snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
   s_write(STDOUT_FILENO, buf, strlen(buf));
534
535
536
537
              argIdx++;
538
539
         snprintf(buf, sizeof(buf), "(%s)n", state);
540
         s_write(STDOUT_FILENO, buf, strlen(buf));
541
542
543
       return NULL;
544 }
```

4.37.1.10 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)

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.

Definition at line 161 of file shell_built_ins.c.

```
161
        char** argv = (char**)arg;
162
                                   // Default signal: term (2)
163
        int sig = 2;
        int start_index = 1; // Start after the "kill" command word.
164
165
       char err_buf[128];
166
       // Check if the first argument specifies a signal
if (argv[start_index] && argv[start_index][0] == '-') {
  if (strcmp(argv[start_index], "-term") == 0) {
167
168
169
170
171
          } else if (strcmp(argv[start_index], "-stop") == 0) {
172
            sig = 0;
          } else if (strcmp(argv[start_index], "-cont") == 0) {
173
174
            sig = 1;
175
          } else {
            // Construct error message
176
177
             s_exit();
178
            return NULL;
179
180
          start index++:
181
182
183
        // Process each PID argument using strtol
184
        for (int i = start_index; argv[i] != NULL; i++) {
185
          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>
186
187
188
            if (s_write(STDERR_FILENO, err_buf, strlen(err_buf)) == -1) {
    u_perror("s_write error");
189
190
191
            continue:
192
193
194
          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);</pre>
195
196
            if (s_write(STDERR_FILENO, err_buf, strlen(err_buf)) == -1) {
  u_perror("s_write error");
197
198
199
200
          }
201
202
       s_exit();
203
       return NULL;
204 }
```

4.37.1.11 u_logout()

Exits the shell and shutsdown PennOS.

Example Usage: logout

Exits the shell and shutsdown PennOS.

Definition at line 549 of file shell_built_ins.c.

```
549 {
550 s_shutdown_pennos();
551 return NULL;
552 }
```

4.37.1.12 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)

Lists all files in the working directory. For extra credit, it should support relative and absolute file paths.

Definition at line 94 of file shell_built_ins.c.

```
94 {
95 ls(arg);
96 s_exit();
97 return NULL;
98 }
```

4.37.1.13 u man()

Lists all available commands.

Example Usage: man

Lists all available commands.

Definition at line 300 of file shell_built_ins.c.

```
300
301
       const char* man_string =
            "cat f1 f2 ... : concatenat
"std in), and writes to std out\n"
302
                                      : concatenates provided files (if none, reads from "
303
                         : sleeps for n seconds\n"
: busy waits indefinitely\n"
304
            "sleep n
305
            "busy
            "echo str
306
                                      : echoes back the input string str\n"
           "touch f1 f2 ...
                                      : lists all files in the working directory\n" : for each file, creates empty file if it doesn't "
307
308
309
            "exist yet, otherwise updates its timestamp\n"
310
                                       : renames f1 to f2 (overwrites f2 if it exists) \n"
```

```
"cp f1 f2
                                      : copies f1 to f2 (overwrites f2 if it exists)\n"
311
           "rm f1 f2 ...
"chmod +_ f1
312
                                      : removes the input list of files\n"
313
                                      : changes f1 permissions to +_ specifications "
           "(+x, +rw, etc)\n"
314
                                     : lists all processes on PennOS, displaying PID, "
315
           "PPID, priority, status, and command name\n"
"kill (-__) pidl pid 2 : sends specified signal (term default) to list "
316
317
318
           "of processes\n"
319
           "nice n command
                                      : spawns a new process for command and sets its \mbox{\tt "}
320
           "priority to n n"
           "nice_pid n pid
321
                                      : adjusts the priority level of an existing "
322
            "process to n\n"
323
                                      : lists all available commands in PennOS\n"
           "man
           "bg
324
                                      : resumes most recently stopped process in "
325
           "background or the one specified by job_id\n"
           "fg : brings most recently stopped or background job "
"to foreground or the one specified by job_id\n"
"jobs : lists all jobs\n"
"logout : exits the shell and shuts down PennOS\n"
326
327
328
           "logout
329
330
           "zombify
                                      : creates a child process that becomes a zombie \n"
331
           "orphanify
                                      : creates a child process that becomes an "
332
           "orphan\n";
333
      s_write(STDOUT_FILENO, man_string, strlen(man_string));
334
335
      return NULL;
336 }
```

4.37.1.14 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

Rename a file. If the dst_file file already exists, overwrite it.

Definition at line 123 of file shell_built_ins.c.

```
123 {
124 mv(arg);
125 s_exit();
126 return NULL;
127 }
```

4.37.1.15 u_nice()

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

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

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

Definition at line 255 of file shell_built_ins.c.

```
256
       char* endptr;
2.57
       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</pre>
258
259
260
261
        return NULL;
262
263
       char* command = ((char**)arg)[2];
264
       void* (*ufunc)(void*) = get_associated_ufunc(command);
265
266
       if (ufunc == NULL) {
267
         return NULL; // no matches, don't spawn
268
269
270
       pid_t new_proc_pid = s_spawn(ufunc, &((char**) arg)[2], 0, 1);
271
      ...._proc_pra != -1) { // non-err
s_nice(new_proc_pid, new_priority);
}
       if (new_proc_pid != -1) { // non-error case
273
274
275
       return NULL;
276
277 }
```

4.37.1.16 u_nice_pid()

Adjust the priority level of an existing process.

Example Usage: nice_pid 0 123 (sets priority 0 to PID 123)

Adjust the priority level of an existing process.

Definition at line 282 of file shell_built_ins.c.

```
282
283
        char* endptr;
        errno = 0;
        int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10);
if (*endptr != '\0' || errno != 0) { // error catch
285
286
287
          return NULL;
288
       pid_t pid = (pid_t)strtol(((char**)arg)[2], &endptr, 10);
if (*endptr != '\0' || errno != 0) {
    return NULL;
289
290
291
292
293
        s_nice(pid, new_priority);
294
       return NULL;
295 }
```

4.37.1.17 u_orphanify()

Used to test orphanifying functionality of your kernel.

Example Usage: orphanify

Used to test orphanifying functionality of your kernel.

Definition at line 591 of file shell_built_ins.c.

4.37.1.18 u_ps()

List all processes on PennOS, displaying PID, PPID, priority, status, and command name.

Example Usage: ps

List all processes on PennOS, displaying PID, PPID, priority, status, and command name.

Definition at line 151 of file shell_built_ins.c.

4.37.1.19 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

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.)

Definition at line 141 of file shell built ins.c.

```
141 {
142 rm(arg);
143 s_exit();
144 return NULL;
```

4.37.1.20 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

Sleep for n seconds.

Definition at line 51 of file shell_built_ins.c.

```
char* endptr;
52
53
     errno = 0;
     if (((char**)arg)[1] == NULL) { // no arg case
54
55
       s_exit();
57
    int sleep_secs = (int)strtol(((char**)arg)[1], &endptr, 10); if (*endptr != '\0' || errno != 0 || sleep_secs <= 0) {
58
59
      s_exit();
60
       return NULL;
61
64
     int sleep_ticks = sleep_secs * 10;
6.5
     s_sleep(sleep_ticks);
66
    s exit();
     return NULL;
```

4.37.1.21 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

For each file, create an empty file if it doesn't exist, else update its timestamp.

Definition at line 114 of file shell built ins.c.

```
114

115 touch(arg);

116 s_exit();

117 return NULL;

118 }
```

4.37.1.22 u_zombify()

Used to test zombifying functionality of your kernel.

Example Usage: zombify

Used to test zombifying functionality of your kernel.

Definition at line 570 of file shell_built_ins.c.

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, 78
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, 59	fs_helpers.c, 64
fs_helpers.h, 74	fs helpers.h, 80
_ ,	ср
block_size	fat_routines.c, 29
fs_helpers.c, 70	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, 257
scheduler.c, 163	shell_built_ins.c, 274
scheduler.h, 174	current_pcbs
child_meta	kern pcb.c, 128
spthread_fwd_args_st, 16	kern_sys_calls.c, 143
child_pcbs	scheduler.c, 170
pcb_st, 13	current_running_pcb
child_with_changed_process_status	fat_routines.c, 37
scheduler.c, 164	fs_kfuncs.c, 100
scheduler.h, 174	kern pcb.c, 128
chmod	kern sys calls.c, 143
fat_routines.c, 28	scheduler.c, 170
fat_routines.h, 45	Scriedulei.c, 170
cmd	data
job_st, 9	vec_st, 21
cmd_str	decrement_fd_ref_count
pcb_st, 13	fs_helpers.c, 65
cmpctdir	fs helpers h 82

delete_from_explicit_queue	cp, <mark>29</mark>
kern_sys_calls.c, 136	current_running_pcb, 37
kern_sys_calls.h, 146	ls, 30
delete_from_queue	mkfs, 31
kern_sys_calls.c, 136	mount, 32
kern_sys_calls.h, 147	mv, 33
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, 39
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	Is, 48
dir entry t, 5	mkfs, 49
firstBlock, 5	mount, 50
mtime, 5	mv, 51
	PERM EXEC, 39
name, 6	PERM NONE, 40
perm, 6	- · · · ·
reserved, 6	PERM_READ, 40
size, 6	PERM_READ_EXEC, 40
type, 6	PERM_READ_WRITE, 40
ele_dtor_fn	PERM_READ_WRITE_EXEC, 40
	PERM_WRITE, 40
vec_st, 21	rm, 53
execute_command	touch, 54
shell.c, 249	TYPE_DIRECTORY, 41
EXITED_NORMALLY	TYPE_REGULAR, 41
signal.h, 183	TYPE_SYMLINK, 41
EXPECT_COMMANDS	TYPE_UNKNOWN, 41
parser.h, 242	unmount, 55
EXPECT_INPUT_FILENAME	fat_size
parser.h, 242	fs_helpers.c, 70
EXPECT_OUTPUT_FILENAME	fs_helpers.h, 87
parser.h, 243	fd_entry_t, 7
E ADDEND	filename, 7
F_APPEND	first_block, 7
fat_routines.h, 39	in_use, 7
F_READ	mode, 7
fat_routines.h, 39	position, 8
F_WRITE	ref_count, 8
fat_routines.h, 39	size, 8
fat	fd_table
fs_helpers.c, 70	fs_helpers.c, 70
fs_helpers.h, 87	fs_helpers.h, 88
FAT_EOF	pcb_st, 13
fat_routines.h, 39	FILE_DESCRIPTOR_TABLE_SIZE
FAT_FREE	kern_pcb.h, 130
fat_routines.h, 39	filename
fat_routines.c	fd_entry_t, 7
cat, 24	fill_buffer_until_full_or_newline
chmod, 28	shell.c, 251
cmpctdir, 29	

find_file	fd_table, 88
fs_helpers.c, 66	find_file, 82
fs_helpers.h, 82	fs_fd, 88
findJobByIdOrCurrent	get_free_fd, 84
shell_built_ins.c, 263	has_executable_permission, 84
shell_built_ins.h, 277	increment_fd_ref_count, 85
FINISHED	init_fd_table, 85
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, 90
shell.c, 251	k_ls, 90
free_pcb	k_lseek, 92
kern pcb.c, 125	k_open, 93
kern pcb.h, 131	k_read, 95
_	
free_scheduler_queues	k_unlink, 96
scheduler.c, 165	k_write, 97
scheduler.h, 176	fs_kfuncs.h
fs_fd	k_close, 102
fs_helpers.c, 70	k_ls, 103
fs_helpers.h, 88	k_lseek, 105
fs_helpers.c	k_open, 106
add_file_entry, 57	k_read, 109
allocate_block, 59	k_unlink, 110
block_size, 70	k_write, 111
compact_directory, 59	SEEK_CUR, 102
copy_host_to_pennfat, 62	SEEK_END, 102
copy_pennfat_to_host, 63	SEEK_SET, 102
copy_source_to_dest, 64	fs_syscalls.c
decrement_fd_ref_count, 65	s_close, 116
fat, 70	s_ls, 116
fat_size, 70	s_lseek, 116
fd_table, 70	s_open, 117
find_file, 66	s_read, 117
fs_fd, 70	s_unlink, 117
get_free_fd, 67	s_write, 118
has_executable_permission, 67	fs_syscalls.h
increment_fd_ref_count, 68	s_close, 120
init_fd_table, 68	s_ls, 120
is_mounted, 70	s_lseek, 121
mark_entry_as_deleted, 69	s_open, 121
MAX FDS, 71	s read, 122
num_fat_blocks, 71	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	
copy_host_to_pennfat, 78	generate_next_priority
copy_pennfat_to_host, 79	scheduler.c, 165
copy_source_to_dest, 80	scheduler.h, 176
decrement_fd_ref_count, 82	get_associated_ufunc
fat, 87	shell_built_ins.c, 274
fat_size, 87	get_free_fd
iαι_5i26, 01	fs_helpers.c, 67
	- <u> </u>

fs_helpers.h, 84	job_state_t, 236
get_next_pcb	RUNNING, 236
scheduler.c, 166	STOPPED, 236
scheduler.h, 177	job_list
get_pcb_in_queue	shell.c, 258
scheduler.c, 166	shell_built_ins.c, 275
scheduler.h, 178	job_st, 8
,	cmd, 9
handle_signal	finished_count, 9
scheduler.c, 167	id, 9
scheduler.h, 178	num_pids, 9
hang	pgid, 9
stress.c, 186	pids, 10
stress.h, 187	state, 10
has_executable_permission	
fs_helpers.c, 67	job_state_t
fs helpers.h, 84	Job.h, 236
10_1101p013.11, 04	JUMP_OUT
id	parser.c, 237
job_st, 9	k_close
in use	fs kfuncs.c, 90
fd_entry_t, 7	fs kfuncs.h, 102
increment_fd_ref_count	_
	k_ls
fs_helpers.c, 68	fs_kfuncs.c, 90
fs_helpers.h, 85	fs_kfuncs.h, 103
init_fd_table	k_lseek
fs_helpers.c, 68	fs_kfuncs.c, 92
fs_helpers.h, 85	fs_kfuncs.h, 105
init_func	k_open
kern_sys_calls.c, 137	fs_kfuncs.c, 93
kern_sys_calls.h, 148	fs_kfuncs.h, 106
initialize_scheduler_queues	k_proc_cleanup
scheduler.c, 167	kern_pcb.c, 126
scheduler.h, 179	kern_pcb.h, 132
input_fd	k_proc_create
pcb_st, 13	kern_pcb.c, 127
input_fd_script	kern_pcb.h, 133
shell.c, 257	k read
is_append	fs_kfuncs.c, 95
shell.c, 257	fs_kfuncs.h, 109
shell.h, 261	k unlink
is_background	fs_kfuncs.c, 96
parsed_command, 11	fs_kfuncs.h, 110
is_file_append	k write
parsed_command, 11	fs_kfuncs.c, 97
is_mounted	fs kfuncs.h, 111
fs_helpers.c, 70	kern_pcb.c
fs_helpers.h, 88	create_pcb, 125
is_sleeping	current_pcbs, 128
_ · · ·	
pcb_st, 13	current_running_pcb, 128
jid_t	free_pcb, 125
Job.h, 236	k_proc_cleanup, 126
job	k_proc_create, 127
	next_pid, 128
Job.h, 236	remove_child_in_parent, 127
Job.h	kern_pcb.h
FINISHED, 236	create_pcb, 130
jid_t, 236	FILE_DESCRIPTOR_TABLE_SIZE, 130
job, 236	free_pcb, 131

k_proc_cleanup, 132	logger.c, 157
k_proc_create, 133	logger.h, 160
pcb_t, 130	logger.c
remove_child_in_parent, 134	log_generic_event, 156
kern_sys_calls.c	log_nice_event, 157
current_fg_pid, 143	log_scheduling_event, 157
current_pcbs, 143	logger.h
current_running_pcb, 143	log_fd, 161
delete_from_explicit_queue, 136	log_generic_event, 158
delete_from_queue, 136	log_nice_event, 160
determine_index_in_queue, 136	log_scheduling_event, 160
init_func, 137	tick_counter, 161
move_pcb_correct_queue, 137	Is
one_priority_queue, 144	fat_routines.c, 30
s_cleanup_init_process, 138	fat_routines.h, 48
s_echo, 138	
s_exit, 139	main
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, 69
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, 71
two_priority_queue, 144	fs_helpers.h, 88
zero_priority_queue, 144	MAX_LINE_BUFFER_SIZE
zombie_queue, 144	shell.c, 249
kern_sys_calls.h	meta
delete_from_explicit_queue, 146	spthread_st, 20
delete_from_queue, 147	meta_mutex
determine_index_in_queue, 147	spthread_meta_st, 17
init_func, 148	MILISEC_IN_NANO
move_pcb_correct_queue, 148	spthread.c, 195
s_cleanup_init_process, 149	mkfs
s_echo, 149	fat_routines.c, 31
s_exit, 150	fat_routines.h, 49
s_kill, 150	mode
s_nice, 151	fd_entry_t, 7
s_ps, 151	mount fat_routines.c, 32
s_sleep, 152	fat routines.h, 50
s_spawn, 153	move_pcb_correct_queue
s_spawn_init, 154	kern sys calls.c, 137
s_waitpid, 154	kern sys calls.h, 148
length	mtime
vec_st, 21	dir_entry_t, 5
log_fd	mv
logger.h, 161	fat_routines.c, 33
pennos.c, 230	fat routines.h, 51
scheduler.c, 171	
log generic event	name
logger.c, 156	dir_entry_t, 6
logger.h, 158	next_job_id
log_nice_event	shell.c, 258
logger.c, 157	next_pid
logger.h, 160	kern_pcb.c, 128
log_scheduling_event	nohang

stress.c, 186	pennos-errno.h, 193
stress.h, 188	P_ERRNO
num_commands	pennos-errno.c, 189
parsed_command, 11	pennos-errno.h, 194
num_fat_blocks	P_ESIGNAL
fs_helpers.c, 71	pennos-errno.h, 193
fs_helpers.h, 88	P_EUNKNOWN
num_pids	pennos-errno.h, 193
job_st, 9	P_EWRITE
one_priority_queue	pennos-errno.h, 193
kern_sys_calls.c, 144	P_INITFAIL
scheduler.c, 171	pennos-errno.h, 193 P NEEDF
orphan_child	pennos-errno.h, 194
shell_built_ins.c, 264	P SIGCONT
output_fd	signal.h, 183
pcb st, 14	P SIGSTOP
output fd script	signal.h, 184
shell.c, 258	P SIGTERM
	signal.h, 184
P_EBADF	P WIFEXITED
pennos-errno.h, 190	signal.h, 184
P_EBUSY	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	parse_command
pennos-errno.h, 190	parser.c, 237
P_EFS_NOT_MOUNTED	parser.h, 244
pennos-errno.h, 190 P EFULL	parsed_command, 10
pennos-errno.h, 191	commands, 11
P EFUNC	is_background, 11
pennos-errno.h, 191	is_file_append, 11
P EINTR	num_commands, 11
pennos-errno.h, 191	stdin_file, 11
P EINVAL	stdout_file, 11
pennos-errno.h, 191	parser.c JUMP_OUT, 237
P_ELSEEK	parse_command, 237
pennos-errno.h, 191	print_parsed_command, 240
P_EMALLOC	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	print_parser_errcode, 247
P_EOPEN	UNEXPECTED_AMPERSAND, 243
pennos-errno.h, 192	UNEXPECTED_FILE_INPUT, 243
P_EPARSE	UNEXPECTED_FILE_OUTPUT, 243
pennos-errno.h, 192	UNEXPECTED_PIPELINE, 243
P_EPERM	pcb_st, 12
pennos-errno.h, 192 P EREAD	child_pcbs, 13
pennos-errno.h, 193	cmd_str, 13
P EREDIR	fd_table, 13
I_LITEDIIT	

input_fd, 13	fat_routines.h, 40
is_sleeping, 13	PERM_READ_WRITE
output_fd, 14	fat_routines.h, 40
par_pid, 14	PERM READ WRITE EXEC
pid, 14	fat_routines.h, 40
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
pennfat.c	position
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
-	remove_child_in_parent
pennos.c	kern_pcb.c, 127
log_fd, 230	kern_pcb.h, 134
main, 230	–
tick_counter, 230	reserved
perm	dir_entry_t, 6
dir_entry_t, 6	rm
PERM_EXEC	fat_routines.c, 34
fat_routines.h, 39	fat_routines.h, 53
PERM_NONE	RUNNING
fat_routines.h, 40	Job.h, 236
PERM_READ	
fat routines.h, 40	s_cleanup_init_process
PERM READ EXEC	kern_sys_calls.c, 138
· · ·- •	kern_sys_calls.h, 149

s_close	curr_priority_arr_index, 170
fs_syscalls.c, 116	current pcbs, 170
fs_syscalls.h, 120	current_running_pcb, 170
s_echo	delete_process_from_all_queues, 164
kern_sys_calls.c, 138	delete_process_from_all_queues_except_current,
kern_sys_calls.h, 149	164
s exit	delete_process_from_particular_queue, 165
kern_sys_calls.c, 139	det_priorities_arr, 171
kern_sys_calls.h, 150	free_scheduler_queues, 165
s_kill	generate_next_priority, 165
kern_sys_calls.c, 139	get_next_pcb, 166
kern_sys_calls.h, 150	get_pcb_in_queue, 166
s_ls	handle_signal, 167
fs_syscalls.c, 116	initialize_scheduler_queues, 167
fs_syscalls.h, 120	log_fd, 171
s_lseek	one_priority_queue, 171
fs_syscalls.c, 116	put_pcb_into_correct_queue, 168
fs_syscalls.h, 121	s_shutdown_pennos, 168
s_nice	scheduler, 168
kern_sys_calls.c, 139	sleep_blocked_queue, 171
kern_sys_calls.h, 151	tick_counter, 171
s_open	two_priority_queue, 171
fs_syscalls.c, 117	zero_priority_queue, 172
fs_syscalls.h, 121	zombie_queue, 172
s_ps	scheduler.h
kern_sys_calls.c, 140	alarm_handler, 173
kern_sys_calls.h, 151	child_in_zombie_queue, 174
s_read	child_with_changed_process_status, 174
fs_syscalls.c, 117	delete_process_from_all_queues, 175
fs_syscalls.h, 122	delete_process_from_all_queues_except_current,
s_shutdown_pennos	175
scheduler.c, 168	delete_process_from_particular_queue, 176
scheduler.h, 180	free_scheduler_queues, 176
s_sleep	generate_next_priority, 176
kern_sys_calls.c, 140	get_next_pcb, 177
kern_sys_calls.h, 152	get_pcb_in_queue, 178
s_spawn	handle_signal, 178
kern_sys_calls.c, 141	initialize_scheduler_queues, 179
kern_sys_calls.h, 153	put_pcb_into_correct_queue, 180
s_spawn_init	s_shutdown_pennos, 180
kern_sys_calls.c, 141	scheduler, 180
kern_sys_calls.h, 154	script_fd
s_unlink	shell.c, 258
fs_syscalls.c, 117	SEEK_CUR
fs_syscalls.h, 123	fs_kfuncs.h, 102
s_waitpid	SEEK_END
kern_sys_calls.c, 142	fs_kfuncs.h, 102
kern_sys_calls.h, 154	SEEK_SET
s_write	fs_kfuncs.h, 102
fs_syscalls.c, 118	setup_cond
fs_syscalls.h, 123	spthread_fwd_args_st, 16
scheduler	setup_done
scheduler.c, 168	spthread_fwd_args_st, 16
scheduler.h, 180	setup_mutex
scheduler.c	spthread_fwd_args_st, 17
alarm_handler, 163	setup_terminal_signal_handlers
child_in_zombie_queue, 163	shell.c, 251
child_with_changed_process_status, 164	shell

shell.c, 252	u_chmod, 279
shell.h, 259	u_cp, 280
shell.c	u_echo, 280
current_fg_pid, 257	u_fg, 281
execute_command, 249	u jobs, 282
fill_buffer_until_full_or_newline, 251	u_kill, 283
free_job_ptr, 251	u_logout, 283
	_ ·
input_fd_script, 257	u_ls, 284
is_append, 257	u_man, 284
job_list, 258	u_mv, 285
MAX_BUFFER_SIZE, 248	u_nice, 285
MAX_LINE_BUFFER_SIZE, 249	u_nice_pid, 286
next_job_id, 258	u_orphanify, 286
output_fd_script, 258	u_ps, 287
PROMPT, 249	u_rm, 287
script_fd, 258	u_sleep, 287
setup terminal signal handlers, 251	u_touch, 288
shell, 252	u zombify, 288
shell sigint handler, 255	shell_sigint_handler
shell_sigstp_handler, 255	shell.c, 255
u execute command, 255	
	shell_sigstp_handler
u_read_and_execute_script, 256	shell.c, 255
shell.h	shutup_mutex
is_append, 261	spthread_signal_args_st, 18
shell, 259	signal
shell_built_ins.c	spthread_signal_args_st, 19
current_fg_pid, 274	signal.h
findJobByldOrCurrent, 263	CONT_BY_SIG, 183
get_associated_ufunc, 274	EXITED_NORMALLY, 183
job_list, 275	P_SIGCONT, 183
orphan_child, 264	P_SIGSTOP, 184
u_bg, 264	P SIGTERM, 184
u_busy, 265	P WIFEXITED, 184
u_cat, 265	P WIFSIGNALED, 184
u_chmod, 266	P WIFSTOPPED, 184
u_cp, 266	STOPPED_BY_SIG, 185
u_echo, 266	TERM BY SIG, 185
u_fg, 267	signals
u_ig, 207 u_iobs, 268	<u> </u>
	pcb_st, 15
u_kill, 269	SIGPTHD
u_logout, 269	spthread.h, 203
u_ls, 270	size
u_man, 270	dir_entry_t, 6
u_mv, 271	fd_entry_t, 8
u_nice, 271	sleep_blocked_queue
u_nice_pid, 271	kern_sys_calls.c, 144
u_orphanify, 272	scheduler.c, 171
u_ps, 272	spthread.c
u rm, 272	_GNU_SOURCE, 195
u_sleep, 273	MILISEC_IN_NANO, 195
u_touch, 273	pthread_fn, 197
u_zombify, 273	spthread_cancel, 197
zombie child, 274	spthread_continue, 197
shell_built_ins.h	spthread_create, 198
findJobByIdOrCurrent, 277	spthread_disable_interrupts_self, 199
u_bg, 278	spthread_enable_interrupts_self, 199
u_busy, 279	spthread_equal, 200
u_cat, 279	spthread_exit, 200

spthread_fwd_args, 197	spthread.c, 200
spthread_join, 200	spthread.h, 207
spthread_meta_t, 197	spthread_meta_st, 17
SPTHREAD_RUNNING_STATE, 196	meta_mutex, 17
spthread_self, 200	state, 17
SPTHREAD_SIG_CONTINUE, 196	suspend_set, 18
SPTHREAD_SIG_SUSPEND, 196	spthread_meta_t
spthread_signal_args, 197	spthread.c, 197
spthread_suspend, 201	spthread.h, 203
	SPTHREAD_RUNNING_STATE
spthread_suspend_self, 201	
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_cancer spthread.c, 197	spthread_suspend
•	•
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, 89
spthread.h, 206	SRC/fs/fs_kfuncs.h, 101
spthread_exit	SRC/fs/fs syscalls.c, 115
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_rwd_args 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, 156
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

SRC/kernel/stress.h, 187	time_to_wake
SRC/lib/pennos-errno.c, 188	pcb_st, 15
SRC/lib/pennos-errno.h, 189	touch
SRC/lib/spthread.c, 194	fat_routines.c, 35
SRC/lib/spthread.h, 202	fat_routines.h, 54
SRC/lib/Vec.c, 209	two_priority_queue
SRC/lib/Vec.h, 217	kern_sys_calls.c, 144
SRC/pennfat.c, 227	scheduler.c, 171
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, 41
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, 258	fat_routines.h, 41
SRC/shell/shell_built_ins.c, 262	TYPE_UNKNOWN
SRC/shell/shell_built_ins.h, 275	fat_routines.h, 41
state	lat_routilles.ii, 41
	u_bg
job_st, 10	shell_built_ins.c, 264
spthread_meta_st, 17 STDERR FILENO	shell_built_ins.h, 278
_	u_busy
fs_syscalls.h, 119	shell built ins.c, 265
stdin_file	shell_built_ins.h, 279
parsed_command, 11	u cat
STDIN_FILENO	shell_built_ins.c, 265
fs_syscalls.h, 119	shell_built_ins.h, 279
stdout_file	
parsed_command, 11	u_chmod
STDOUT_FILENO	shell_built_ins.c, 266
fs_syscalls.h, 119	shell_built_ins.h, 279
STOPPED	u_cp
Job.h, 236	shell_built_ins.c, 266
STOPPED_BY_SIG	shell_built_ins.h, 280
signal.h, 185	u_echo
stress.c	shell_built_ins.c, 266
crash, 186	shell_built_ins.h, 280
hang, 186	u_execute_command
nohang, 186	shell.c, 255
recur, 186	u_fg
stress.h	shell_built_ins.c, 267
crash, 187	shell_built_ins.h, 281
hang, 187	u_jobs
nohang, 188	shell_built_ins.c, 268
recur, 188	shell_built_ins.h, 282
recur, 188 suspend set	shell_built_ins.h, 282 u_kill
suspend_set	
	u_kill
suspend_set	u_kill shell_built_ins.c, 269
suspend_set spthread_meta_st, 18	u_kill shell_built_ins.c, 269 shell_built_ins.h, 283
suspend_set spthread_meta_st, 18 TERM_BY_SIG	u_kill shell_built_ins.c, 269 shell_built_ins.h, 283 u_logout
suspend_set spthread_meta_st, 18 TERM_BY_SIG signal.h, 185	u_kill shell_built_ins.c, 269 shell_built_ins.h, 283 u_logout shell_built_ins.c, 269
suspend_set spthread_meta_st, 18 TERM_BY_SIG signal.h, 185 thread	u_kill shell_built_ins.c, 269 shell_built_ins.h, 283 u_logout shell_built_ins.c, 269 shell_built_ins.h, 283
suspend_set spthread_meta_st, 18 TERM_BY_SIG signal.h, 185 thread spthread_st, 20	u_kill shell_built_ins.c, 269 shell_built_ins.h, 283 u_logout shell_built_ins.c, 269 shell_built_ins.h, 283 u_ls
suspend_set spthread_meta_st, 18 TERM_BY_SIG signal.h, 185 thread spthread_st, 20 thread_handle	u_kill shell_built_ins.c, 269 shell_built_ins.h, 283 u_logout shell_built_ins.c, 269 shell_built_ins.h, 283 u_ls shell_built_ins.c, 270
suspend_set spthread_meta_st, 18 TERM_BY_SIG signal.h, 185 thread spthread_st, 20 thread_handle pcb_st, 15 tick_counter	u_kill shell_built_ins.c, 269 shell_built_ins.h, 283 u_logout shell_built_ins.c, 269 shell_built_ins.h, 283 u_ls shell_built_ins.c, 270 shell_built_ins.h, 284 u_man
suspend_set spthread_meta_st, 18 TERM_BY_SIG signal.h, 185 thread spthread_st, 20 thread_handle pcb_st, 15 tick_counter kern_sys_calls.c, 144	u_kill shell_built_ins.c, 269 shell_built_ins.h, 283 u_logout shell_built_ins.c, 269 shell_built_ins.h, 283 u_ls shell_built_ins.c, 270 shell_built_ins.h, 284 u_man shell_built_ins.c, 270
suspend_set spthread_meta_st, 18 TERM_BY_SIG signal.h, 185 thread spthread_st, 20 thread_handle pcb_st, 15 tick_counter kern_sys_calls.c, 144 logger.h, 161	u_kill shell_built_ins.c, 269 shell_built_ins.h, 283 u_logout shell_built_ins.c, 269 shell_built_ins.h, 283 u_ls shell_built_ins.c, 270 shell_built_ins.h, 284 u_man shell_built_ins.c, 270 shell_built_ins.h, 284
suspend_set spthread_meta_st, 18 TERM_BY_SIG signal.h, 185 thread spthread_st, 20 thread_handle pcb_st, 15 tick_counter kern_sys_calls.c, 144	u_kill shell_built_ins.c, 269 shell_built_ins.h, 283 u_logout shell_built_ins.c, 269 shell_built_ins.h, 283 u_ls shell_built_ins.c, 270 shell_built_ins.h, 284 u_man shell_built_ins.c, 270

shell_built_ins.h, 285	Vec, 219
u_nice	vec_capacity, 218
shell_built_ins.c, 271	vec_clear, 219
shell_built_ins.h, 285	vec_destroy, 220
u_nice_pid	vec_erase, 220
shell_built_ins.c, 271	vec_erase_no_deletor, 221
shell_built_ins.h, 286	vec_get, 222
u_orphanify	vec_insert, 222
shell_built_ins.c, 272	vec_is_empty, 218
shell_built_ins.h, 286	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, 272	vec_set, 226
shell_built_ins.h, 287	vec_capacity
u_read_and_execute_script	Vec.h, 218
shell.c, 256	vec_clear
u_rm	Vec.c, 209
shell_built_ins.c, 272	Vec.h, 219
shell_built_ins.h, 287	vec_destroy
u_sleep	Vec.c, 210
shell_built_ins.c, 273	Vec.h, 220
shell_built_ins.h, 287	vec_erase
u_touch	Vec.c, 210
shell_built_ins.c, 273	Vec.h, 220
shell_built_ins.h, 288	vec_erase_no_deletor Vec.c, 211
u_zombify	
shell_built_ins.c, 273	Vec.h, 221
shell_built_ins.h, 288 UNEXPECTED AMPERSAND	vec_get Vec.c, 212
parser.h, 243	Vec.h, 222
UNEXPECTED FILE INPUT	vec insert
parser.h, 243	Vec_insert Vec.c, 212
UNEXPECTED FILE OUTPUT	Vec.b, 222
parser.h, 243	vec_is_empty
UNEXPECTED_PIPELINE	Vec.h, 218
parser.h, 243	vec len
unmount	Vec.h, 218
fat routines.c, 36	vec_new
fat routines.h, 55	Vec.c, 213
	Vec.h, 223
Vec	vec_pop_back
Vec.h, 219	Vec.c, 214
Vec.c	Vec.h, 224
vec_clear, 209	vec_push_back
vec_destroy, 210	Vec.c, 214
vec_erase, 210	Vec.h, 224
vec_erase_no_deletor, 211	vec_resize
vec_get, 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 zero_priority_queue kern_sys_calls.c, 144 scheduler.c, 172 zombie_child shell_built_ins.c, 274 zombie_queue kern_sys_calls.c, 144 scheduler.c, 172