

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	11
3.4.2.1 commands	11
3.4.2.2 is_background	11
3.4.2.3 is_file_append	11

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

3.9.1 Detailed Description	19
3.9.2 Member Data Documentation	20
3.9.2.1 meta	20
3.9.2.2 thread	20
3.10 vec_st Struct Reference	20
3.10.1 Detailed Description	20
3.10.2 Member Data Documentation	20
3.10.2.1 capacity	21
3.10.2.2 data	21
3.10.2.3 ele_dtor_fn	21
3.10.2.4 length	21
4 File Documentation	23
4.1 SRC/fs/fat_routines.c File Reference	23
4.1.1 Function Documentation	24
4.1.1.1 cat()	24
4.1.1.2 chmod()	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()	105
4.6.2.4 k_open()	107
4.6.2.5 k_read()	109
4.6.2.6 k_unlink()	110
4.6.2.7 k_write()	112
4.7 SRC/fs/fs_syscalls.c File Reference	115
4.7.1 Function Documentation	116
4.7.1.1 s_close()	116
4.7.1.2 s_ls()	116
4.7.1.3 s_lseek()	117
4.7.1.4 s_open()	117
4.7.1.5 s_read()	117
4.7.1.6 s_unlink()	118
4.7.1.7 s_write()	118
4.8 SRC/fs/fs_syscalls.h File Reference	118
4.8.1 Macro Definition Documentation	119
4.8.1.1 STDERR_FILENO	119
4.8.1.2 STDIN_FILENO	119
4.8.1.3 STDOUT_FILENO	119
4.8.2 Function Documentation	120
4.8.2.1 s_close()	120
4.8.2.2 s_ls()	120
4.8.2.3 s_lseek()	121
4.8.2.4 s_open()	122
4.8.2.5 s_read()	122
4.8.2.6 s_unlink()	123
4.8.2.7 s_write()	124
4.9 SRC/kernel/kern_pcb.c File Reference	124
4.9.1 Function Documentation	125
4.9.1.1 create_pcb()	125
4.9.1.2 free_pcb()	126
4.9.1.3 k_proc_cleanup()	126
4.9.1.4 k_proc_create()	127
4.9.1.5 remove_child_in_parent()	128
4.9.2 Variable Documentation	128
4.9.2.1 current_pcb	128
4.9.2.2 current_running_pcb	128
4.9.2.3 next_pid	128
4.10 SRC/kernel/kern_pcb.h File Reference	129
4.10.1 Macro Definition Documentation	130
4.10.1.1 FILE_DESCRIPTOR_TABLE_SIZE	130
4.10.2 Typedef Documentation	130

4.10.2.1 pcb_t	130
4.10.3 Function Documentation	130
4.10.3.1 create_pcb()	130
4.10.3.2 free_pcb()	131
4.10.3.3 k_proc_cleanup()	132
4.10.3.4 k_proc_create()	133
4.10.3.5 remove_child_in_parent()	134
4.11 SRC/kernel/kern_sys_calls.c File Reference	134
4.11.1 Function Documentation	136
4.11.1.1 delete_from_explicit_queue()	136
4.11.1.2 delete_from_queue()	136
4.11.1.3 determine_index_in_queue()	137
4.11.1.4 init_func()	137
4.11.1.5 move_pcb_correct_queue()	137
4.11.1.6 s_cleanup_init_process()	138
4.11.1.7 s_echo()	138
4.11.1.8 s_exit()	139
4.11.1.9 s_kill()	139
4.11.1.10 s_nice()	140
4.11.1.11 s_ps()	140
4.11.1.12 s_sleep()	141
4.11.1.13 s_spawn()	141
4.11.1.14 s_spawn_init()	142
4.11.1.15 s_waitpid()	142
4.11.2 Variable Documentation	143
4.11.2.1 current_fg_pid	143
4.11.2.2 current_pcb	143
4.11.2.3 current_running_pcb	144
4.11.2.4 one_priority_queue	144
4.11.2.5 sleep_blocked_queue	144
4.11.2.6 tick_counter	144
4.11.2.7 two_priority_queue	144
4.11.2.8 zero_priority_queue	144
4.11.2.9 zombie_queue	145
4.12 SRC/kernel/kern_sys_calls.h File Reference	145
4.12.1 Function Documentation	146
4.12.1.1 delete_from_explicit_queue()	146
4.12.1.2 delete_from_queue()	147
4.12.1.3 determine_index_in_queue()	147
4.12.1.4 init_func()	148
4.12.1.5 move_pcb_correct_queue()	148
4.12.1.6 s_cleanup_init_process()	149

4.12.1.7 s_echo()	149
4.12.1.8 s_exit()	150
4.12.1.9 s_kill()	150
4.12.1.10 s_nice()	151
4.12.1.11 s_ps()	152
4.12.1.12 s_sleep()	152
4.12.1.13 s_spawn()	153
4.12.1.14 s_spawn_init()	154
4.12.1.15 s_waitpid()	154
4.13 SRC/kernel/logger.c File Reference	156
4.13.1 Function Documentation	156
4.13.1.1 log_generic_event()	156
4.13.1.2 log_nice_event()	157
4.13.1.3 log_scheduling_event()	158
4.14 SRC/kernel/logger.h File Reference	158
4.14.1 Function Documentation	158
4.14.1.1 log_generic_event()	159
4.14.1.2 log_nice_event()	160
4.14.1.3 log_scheduling_event()	160
4.14.2 Variable Documentation	161
4.14.2.1 log_fd	161
4.14.2.2 tick_counter	161
4.15 SRC/kernel/scheduler.c File Reference	162
4.15.1 Function Documentation	163
4.15.1.1 alarm_handler()	163
4.15.1.2 child_in_zombie_queue()	164
4.15.1.3 child_with_changed_process_status()	164
4.15.1.4 delete_process_from_all_queues()	164
4.15.1.5 delete_process_from_all_queues_except_current()	165
4.15.1.6 delete_process_from_particular_queue()	165
4.15.1.7 free_scheduler_queues()	165
4.15.1.8 generate_next_priority()	166
4.15.1.9 get_next_pcb()	166
4.15.1.10 get_pcb_in_queue()	167
4.15.1.11 handle_signal()	167
4.15.1.12 initialize_scheduler_queues()	168
4.15.1.13 put_pcb_into_correct_queue()	168
4.15.1.14 s_shutdown_pennos()	168
4.15.1.15 scheduler()	169
4.15.2 Variable Documentation	170
4.15.2.1 curr_priority_arr_index	170
4.15.2.2 current_pcb	170

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()	186
4.19.1.3 nohang()	186
4.19.1.4 recur()	187
4.20 SRC/kernel/stress.h File Reference	187
4.20.1 Function Documentation	187
4.20.1.1 crash()	187
4.20.1.2 hang()	188
4.20.1.3 nohang()	188
4.20.1.4 recur()	188
4.21 SRC/lib/pennos-errno.c File Reference	188
4.21.1 Variable Documentation	189
4.21.1.1 P_ERRNO	189
4.22 SRC/lib/pennos-errno.h File Reference	189
4.22.1 Macro Definition Documentation	190
4.22.1.1 P_EBADF	190
4.22.1.2 P_EBUSY	190
4.22.1.3 P_ECLOSE	190
4.22.1.4 P_ECOMMAND	190
4.22.1.5 P_EEXIST	190
4.22.1.6 P_EFS_NOT_MOUNTED	191
4.22.1.7 P_EFULL	191
4.22.1.8 P_EFUNC	191
4.22.1.9 P_EINTR	191
4.22.1.10 P_EINVAL	191
4.22.1.11 P_ELSEEK	191
4.22.1.12 P_EMALLOC	192
4.22.1.13 P_EMAP	192
4.22.1.14 P_ENOENT	192
4.22.1.15 P_ENULL	192
4.22.1.16 P_EOPEN	192
4.22.1.17 P_EPARSE	192
4.22.1.18 P_EPERM	193
4.22.1.19 P_EREAD	193
4.22.1.20 P_EREDIR	193
4.22.1.21 P_ESIGNAL	193
4.22.1.22 P_EUNKNOWN	193
4.22.1.23 P_EWRITE	193
4.22.1.24 P_INITFAIL	194
4.22.1.25 P_NEEDF	194
4.22.2 Variable Documentation	194
4.22.2.1 P_ERRNO	194
4.23 SRC/lib/spthread.c File Reference	194

4.23.1 Macro Definition Documentation	195
4.23.1.1 _GNU_SOURCE	195
4.23.1.2 MILLISEC_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()	206
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()	208

4.24.3.11 <code>spthread_suspend_self()</code>	208
4.25 SRC/lib/Vec.c File Reference	209
4.25.1 Function Documentation	209
4.25.1.1 <code>vec_clear()</code>	209
4.25.1.2 <code>vec_destroy()</code>	210
4.25.1.3 <code>vec_erase()</code>	211
4.25.1.4 <code>vec_erase_no_deletor()</code>	211
4.25.1.5 <code>vec_get()</code>	212
4.25.1.6 <code>vec_insert()</code>	212
4.25.1.7 <code>vec_new()</code>	213
4.25.1.8 <code>vec_pop_back()</code>	214
4.25.1.9 <code>vec_push_back()</code>	214
4.25.1.10 <code>vec_resize()</code>	215
4.25.1.11 <code>vec_set()</code>	216
4.26 SRC/lib/Vec.h File Reference	217
4.26.1 Macro Definition Documentation	218
4.26.1.1 <code>vec_capacity</code>	218
4.26.1.2 <code>vec_is_empty</code>	218
4.26.1.3 <code>vec_len</code>	219
4.26.2 Typedef Documentation	219
4.26.2.1 <code>ptr_dtor_fn</code>	219
4.26.2.2 <code>ptr_t</code>	219
4.26.2.3 <code>Vec</code>	219
4.26.3 Function Documentation	219
4.26.3.1 <code>vec_clear()</code>	219
4.26.3.2 <code>vec_destroy()</code>	220
4.26.3.3 <code>vec_erase()</code>	221
4.26.3.4 <code>vec_erase_no_deletor()</code>	221
4.26.3.5 <code>vec_get()</code>	222
4.26.3.6 <code>vec_insert()</code>	222
4.26.3.7 <code>vec_new()</code>	223
4.26.3.8 <code>vec_pop_back()</code>	224
4.26.3.9 <code>vec_push_back()</code>	224
4.26.3.10 <code>vec_resize()</code>	225
4.26.3.11 <code>vec_set()</code>	226
4.27 SRC/pennfat.c File Reference	227
4.27.1 Macro Definition Documentation	227
4.27.1.1 <code>PROMPT</code>	227
4.27.2 Function Documentation	227
4.27.2.1 <code>main()</code>	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	249

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

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

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

SRC/ pennfat.c	227
SRC/ pennos.c	229
SRC/fs/ fat_routines.c	23
SRC/fs/ fat_routines.h	37
SRC/fs/ fs_helpers.c	56
SRC/fs/ fs_helpers.h	71
SRC/fs/ fs_kfuncs.c	89
SRC/fs/ fs_kfuncs.h	101
SRC/fs/ fs_syscalls.c	115
SRC/fs/ fs_syscalls.h	118
SRC/kernel/ kern_pcb.c	124
SRC/kernel/ kern_pcb.h	129
SRC/kernel/ kern_sys_calls.c	134
SRC/kernel/ kern_sys_calls.h	145
SRC/kernel/ logger.c	156
SRC/kernel/ logger.h	158
SRC/kernel/ scheduler.c	162
SRC/kernel/ scheduler.h	172
SRC/kernel/ signal.c	182
SRC/kernel/ signal.h	183
SRC/kernel/ stress.c	185
SRC/kernel/ stress.h	187
SRC/lib/ pennos-errno.c	188
SRC/lib/ pennos-errno.h	189
SRC/lib/ spthread.c	194
SRC/lib/ spthread.h	202
SRC/lib/ Vec.c	209
SRC/lib/ Vec.h	217
SRC/shell/ builtins.c	231
SRC/shell/ builtins.h	233
SRC/shell/ Job.h	234
SRC/shell/ parser.c	236
SRC/shell/ parser.h	241
SRC/shell/ shell.c	247
SRC/shell/ shell.h	258
SRC/shell/ shell_built_ins.c	262
SRC/shell/ shell_built_ins.h	275

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.

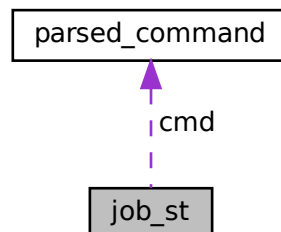
The documentation for this struct was generated from the following file:

- 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:

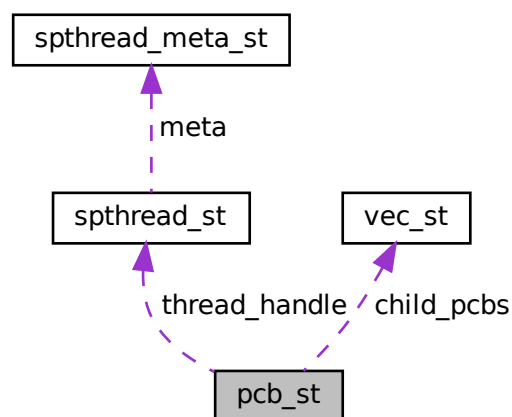
- SRC/shell/[parser.h](#)

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

- [sptthread_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
```

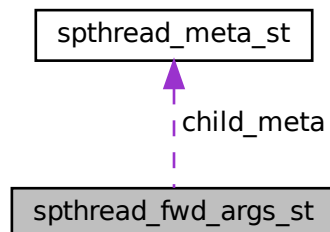
Definition at line 58 of file `kern_pcb.h`.

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`

```
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

```
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

```
volatile sig_atomic_t spthread_signal_args_st::ack
```

Definition at line 48 of file spthread.c.

3.8.2.2 `shutup_mutex`

```
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`.

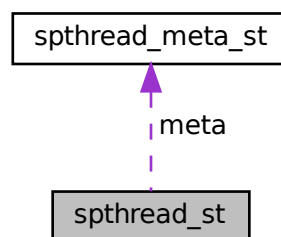
The documentation for this struct was generated from the following file:

- 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`.

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](#)

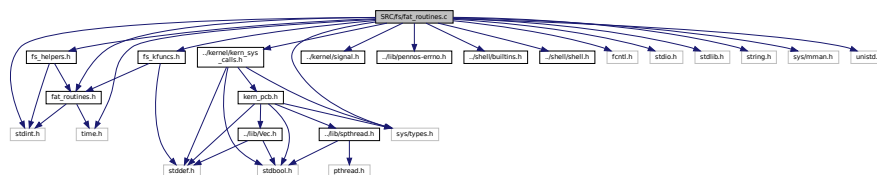
Chapter 4

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 * [ls](#) (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

<i>arg</i>	Arguments array (command line arguments)
------------	--

Returns

void pointer (unused)

Definition at line 214 of file fat_routines.c.

```

214     {
215     char** args = (char**)arg;
216
217     // verify that the file system is mounted
218     if (!is_mounted) {
219         P_ERRNO = P_EFS_NOT_MOUNTED;
220         u_perror("cat");
221         return NULL;
222     }
223
224     // early return if there is nothing after cat
225     if (args[1] == NULL) {
226         // if none of the above conditions, then check if we need to redirect stdin
227         if (current_running_pcb) {
228             // open new stdin
229             int in_fd = current_running_pcb->input_fd;
230             int out_fd = current_running_pcb->output_fd;
231             char* file_1 = fd_table[in_fd].filename;
232             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
236             if ((strcmp(file_1, file_2) == 0) && is_append) {
237                 P_ERRNO = P_EREDIR;
238                 u_perror("cat");
239                 return NULL;
240             }
241
242             // edge case when input and output files names are the same but we're not
243             // appending truncates the file
244             if ((strcmp(file_1, file_2) == 0)) {
245                 return NULL;
246             }
247
248             // get the size of stdin file
249             off_t in_fd_size = k_lseek(in_fd, 0, SEEK_END);
250             if (in_fd_size == -1) {
251                 k_close(in_fd);
252                 u_perror("cat");
253                 return NULL;
254             }
255             if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
256                 k_close(in_fd);
257                 u_perror("cat");
258                 return NULL;
259             }
260
261             char* buffer = (char*)malloc(block_size);
262             if (buffer == NULL) {
263                 P_ERRNO = P_EMALLOC;
264                 k_close(in_fd);
265                 u_perror("cat");
266                 return NULL;
267             }
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 =
274                     bytes_remaining < block_size ? bytes_remaining : block_size;
275                 bytes_read = k_read(in_fd, buffer, bytes_to_read);
276
277                 if (bytes_read <= 0) {
278                     break;
279                 }
280
281                 if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
282                     free(buffer);
283                     k_close(in_fd);
284                     u_perror("cat");

```

```

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

```

```
372         return NULL;
373     }
374 }
375
376 if (out_fd != STDOUT_FILENO) {
377     k_close(out_fd);
378 }
379 return NULL;
380 }
381
382 // handle concatenating one or more files: cat FILE ... [-w/-a OUTPUT_FILE]
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
391 for (i = start; i <= end; i++) {
392     // skip the redirection flags and their arguments
393     if (strcmp(args[i], "-w") == 0 || strcmp(args[i], "-a") == 0) {
394         i++;
395         continue;
396     }
397
398     // open the current input file
399     int in_fd = k_open(args[i], F_READ);
400     if (in_fd < 0) {
401         u_perror("cat");
402         continue;
403     }
404
405     // use lseek to get the size of in_fd
406     off_t in_fd_size = k_lseek(in_fd, 0, SEEK_END);
407     if (in_fd_size == -1) {
408         k_close(in_fd);
409         u_perror("cat");
410         continue;
411     }
412
413     // use lseek to reset position to 0 for reading
414     if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
415         k_close(in_fd);
416         u_perror("cat");
417         continue;
418     }
419
420     // copy file content to output
421     char* buffer = (char*)malloc(block_size);
422     if (buffer == NULL) {
423         P_ERRNO = P_EMALLOC;
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;
435         bytes_read = k_read(in_fd, buffer, bytes_to_read);
436
437         if (bytes_read <= 0) {
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
451     // read error
452     if (bytes_read < 0) {
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 }

```

4.1.1.2 chmod()

```

void* chmod (
    void * arg )

```

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.

```

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

```

```

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

```

4.1.1.3 cmpctdir()

```

void* cmpctdir (
    void * arg )

```

Implements compaction of root directory.

Compacts the root directory by removing all deleted entries.

Definition at line 851 of file fat_routines.c.

```

851     {
852     if (!is_mounted) {
853         P_ERRNO = P_EFS_NOT_MOUNTED;
854         u_perror("cmpctdir");
855         return NULL;
856     }
857
858     if (compact_directory() != 0) {
859         u_perror("cmpctdir");
860     }
861
862     return NULL;
863 }

```

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.

```

640     {

```

```

641 char** args = (char**)arg;
642
643 // check that we have enough arguments
644 if (args[1] == NULL || args[2] == NULL) {
645     P_ERRNO = P_EINVAL;
646     u_perror("cp");
647     return NULL;
648 }
649
650 // cp -h SOURCE DEST
651 if (strcmp(args[1], "-h") == 0) {
652     if (args[2] == NULL || args[3] == NULL) {
653         P_ERRNO = P_EINVAL;
654         u_perror("cp");
655         return NULL;
656     }
657
658     if (copy_host_to_pennfat(args[2], args[3]) != 0) {
659         u_perror("cp");
660         return NULL;
661     }
662     return NULL;
663 }
664
665 // cp SOURCE -h DEST
666 if (args[2] != NULL && strcmp(args[2], "-h") == 0) {
667     if (args[3] == NULL) {
668         P_ERRNO = P_EINVAL;
669         u_perror("cp");
670         return NULL;
671     }
672
673     if (copy_pennfat_to_host(args[1], args[3]) != 0) {
674         u_perror("cp");
675         return NULL;
676     }
677     return NULL;
678 }
679
680 // cp SOURCE DEST
681 if ((args[1] != NULL && strcmp(args[1], "-h") != 0) &&
682     (args[2] != NULL && strcmp(args[2], "-h") != 0) && args[3] == NULL) {
683     if (copy_source_to_dest(args[1], args[2]) != 0) {
684         u_perror("cp");
685         return NULL;
686     }
687     return NULL;
688 }
689
690 P_ERRNO = P_EUNKNOWN;
691 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");
483             return NULL;
484         }

```



```

485     } else {
486         if (k_ls(NULL) == -1) {
487             u_perror("ls");
488             return NULL;
489         }
490     }
491     return NULL;
492 }

```

4.1.1.6 mkfs()

```

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.

Creates a PennFAT filesystem in the file named `fs_name`.

Definition at line 34 of file `fat_routines.c`.

```

34                                     {
35     // validate arguments
36     if (num_blocks < 1 || num_blocks > 32) {
37         P_ERRNO = P_EINVAL;
38         return -1;
39     }
40     if (blk_size < 0 || blk_size > 4) {
41         P_ERRNO = P_EINVAL;
42         return -1;
43     }
44
45     // determine the file system size
46     int block_sizes[] = {256, 512, 1024, 2048, 4096};
47     int actual_block_size = block_sizes[blk_size];
48     int fat_size = num_blocks * actual_block_size;
49     int fat_entries = fat_size / 2;
50     int num_data_blocks =
51         (num_blocks == 32)
52         ? fat_entries - 2
53         : fat_entries - 1; // note: first entry is reserved for metadata!
54     size_t filesystem_size = fat_size + (actual_block_size * num_data_blocks);
55
56     // create the file for the filesystem
57     int fd = open(fs_name, O_RDWR | O_CREAT | O_TRUNC, 0644);
58     if (fd == -1) {
59         P_ERRNO = P_EOPEN;
60         return -1;
61     }
62
63     // extend the file to the required size
64     if (ftruncate(fd, filesystem_size) == -1) {
65         P_ERRNO = P_EFUNC;
66         close(fd);
67         return -1;
68     }
69
70     // allocate the FAT
71     uint16_t* temp_fat = (uint16_t*)calloc(fat_entries, sizeof(uint16_t));
72     if (!temp_fat) {
73         P_ERRNO = P_EMALLOC;
74         close(fd);
75         return -1;
76     }
77
78     // initialize FAT entries to their correct values
79     temp_fat[0] = (num_blocks << 8) | blk_size;
80     temp_fat[1] = FAT_EOF;
81     for (int i = 2; i < fat_entries; i++) {
82         temp_fat[i] = FAT_FREE;
83     }
84
85     // 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
93 // initialize the root directory + write to memory
94 uint8_t* root_dir = (uint8_t*)calloc(actual_block_size, 1);
95 if (lseek(fd, fat_size, SEEK_SET) == -1) {
96     P_ERRNO = P_ELSEEK;
97     free(temp_fat);
98     free(root_dir);
99     close(fd);
100    return -1;
101 }
102 if (write(fd, root_dir, actual_block_size) != actual_block_size) {
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;
115 }

```

4.1.1.7 mount()

```

int mount (
    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     {
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);
129     if (fs_fd == -1) {
130         P_ERRNO = P_ENOENT;
131         return -1;
132     }
133
134     // read the first two bytes to get size configuration
135     uint16_t config;
136     if (read(fs_fd, &config, sizeof(config)) != sizeof(config)) {
137         P_ERRNO = P_EREAD;
138         close(fs_fd);
139         fs_fd = -1;
140         return -1;
141     }
142
143     // extract FAT region size information
144     num_fat_blocks = (config >> 8) & 0xFF; // MSB
145     int block_size_config = config & 0xFF; // LSB
146     int block_sizes[] = {256, 512, 1024, 2048, 4096};
147     block_size = block_sizes[block_size_config];
148     fat_size = num_fat_blocks * block_size;
149
150     // map the FAT region into memory
151     if (lseek(fs_fd, 0, SEEK_SET) == -1) {
152         P_ERRNO = P_ELSEEK;
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);
162     fs_fd = -1;
163     return -1;
164 }
165
166 init_fd_table(fd_table); // initialize the file descriptor table
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

<i>arg</i>	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) {
566         P_ERRNO = P_EFS_NOT_MOUNTED;
567         u_perror("mv");
568         return NULL;
569     }
570
571     // check if we have both source and destination arguments
572     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         return NULL;
584     }
585
586     // check if source file exists
587     dir_entry_t source_entry;
588     int source_offset = find_file(source, &source_entry);
589     if (source_offset < 0) {
590         u_perror("mv");
591         return NULL;
592     }

```

```

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

<i>arg</i>	Arguments array (command line arguments)
------------	--

Returns

void pointer (unused)

Definition at line 698 of file fat_routines.c.

```

698 {
699     char** args = (char**)arg;
700
701     // verify that the file system is mounted
702     if (!is_mounted) {
703         P_ERRNO = P_EFS_NOT_MOUNTED;

```

```

704     u_perror("rm");
705     return NULL;
706 }
707
708 // check if we have any arguments
709 if (args[1] == NULL) {
710     P_ERRNO = P_EINVAL;
711     u_perror("rm");
712     return NULL;
713 }
714
715 // process each file argument
716 for (int i = 1; args[i] != NULL; i++) {
717     // find the file in the directory
718     dir_entry_t entry;
719     int entry_offset = find_file(args[i], &entry);
720
721     if (entry_offset < 0) {
722         // file doesn't exist
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++) {
730         if (fd_table[j].in_use && strcmp(fd_table[j].filename, args[i]) == 0) {
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) {
739         P_ERRNO = P_ELSEEK;
740         u_perror("rm");
741         continue;
742     }
743
744     char deleted = 1; // mark as deleted
745     if (write(fs_fd, &deleted, sizeof(deleted)) != sizeof(deleted)) {
746         P_ERRNO = P_EWRITE;
747         u_perror("rm");
748         continue;
749     }
750
751     // free the FAT chain for this file
752     uint16_t block = entry.firstBlock;
753     while (block != FAT_FREE && block != FAT_EOF) {
754         uint16_t next_block = fat[block];
755         fat[block] = FAT_FREE;
756         block = next_block;
757     }
758 }
759
760 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) {
506         P_ERRNO = P_EFS_NOT_MOUNTED;
507         u_perror("touch");
508         return NULL;
509     }
510
511     // check if we have any arguments
512     if (args[1] == NULL) {
513         P_ERRNO = P_EINVAL;
514         u_perror("touch");
515         return NULL;
516     }
517
518     // process each file argument
519     for (int i = 1; args[i] != NULL; i++) {
520         dir_entry_t entry;
521         int entry_offset = find_file(args[i], &entry);
522
523         // file exists
524         if (entry_offset >= 0) {
525             entry.mtime = time(NULL);
526
527             // write the updated entry back to the directory
528             if (lseek(fs_fd, entry_offset, SEEK_SET) == -1) {
529                 P_ERRNO = P_ELSEEK;
530                 u_perror("touch");
531                 continue;
532             }
533             if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
534                 P_ERRNO = P_EWRITE;
535                 u_perror("touch");
536                 continue;
537             }
538         } else {
539             // file doesn't exist, create a new empty file
540
541             // check if the fat is full
542             if (P_ERRNO == P_EFULL) {
543                 u_perror("touch");
544                 return NULL;
545             }
546
547             // add the file entry to root directory
548             if (add_file_entry(args[i], 0, 0, TYPE_REGULAR, PERM_READ_WRITE) == -1) {
549                 u_perror("touch");
550                 continue;
551             }
552         }
553     }
554     return NULL;
555 }

```

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
181     // unmap the FAT
182     if (fat != NULL) {
183         if (munmap(fat, fat_size) == -1) {
184             P_ERRNO = P_EMAP;
185             return -1;
186         }
187         fat = NULL;
188     }

```

```

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

```

4.1.2 Variable Documentation

4.1.2.1 current_running_pcb

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

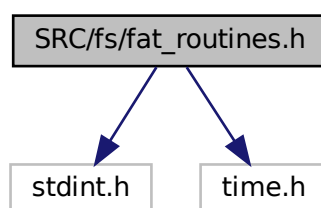
Definition at line 38 of file scheduler.c.

4.2 SRC/fs/fat_routines.h File Reference

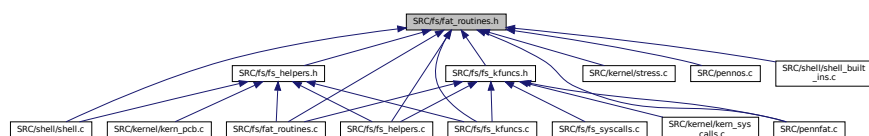
```

#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 * [ls](#) (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

<i>arg</i>	Arguments array (command line arguments)
------------	--

Returns

void pointer (unused)

Definition at line 214 of file fat_routines.c.

```

214     {
215     char** args = (char**)arg;
216
217     // verify that the file system is mounted
218     if (!is_mounted) {
219         P_ERRNO = P_EFS_NOT_MOUNTED;
220         u_perror("cat");
221         return NULL;
222     }
223
224     // early return if there is nothing after cat
225     if (args[1] == NULL) {
226         // if none of the above conditions, then check if we need to redirect stdin
227         if (current_running_pcb) {
228             // open new stdin
229             int in_fd = current_running_pcb->input_fd;
230             int out_fd = current_running_pcb->output_fd;
231             char* file_1 = fd_table[in_fd].filename;
232             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
236             if ((strcmp(file_1, file_2) == 0) && is_append) {
237                 P_ERRNO = P_EREDIR;
238                 u_perror("cat");
239                 return NULL;
240             }
241
242             // edge case when input and output files names are the same but we're not
243             // appending truncates the file
244             if ((strcmp(file_1, file_2) == 0)) {
245                 return NULL;
246             }
247
248             // get the size of stdin file
249             off_t in_fd_size = k_lseek(in_fd, 0, SEEK_END);
250             if (in_fd_size == -1) {
251                 k_close(in_fd);
252                 u_perror("cat");
253                 return NULL;
254             }
255             if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
256                 k_close(in_fd);
257                 u_perror("cat");
258                 return NULL;
259             }
260
261             char* buffer = (char*)malloc(block_size);
262             if (buffer == NULL) {
263                 P_ERRNO = P_EMALLO;
264                 k_close(in_fd);
265                 u_perror("cat");
266                 return NULL;
267             }
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 =
274                     bytes_remaining < block_size ? bytes_remaining : block_size;
275                 bytes_read = k_read(in_fd, buffer, bytes_to_read);
276
277                 if (bytes_read <= 0) {
278                     break;
279                 }
280
281                 if (k_write(out_fd, buffer, bytes_read) != bytes_read) {
282                     free(buffer);
283                     k_close(in_fd);
284                     u_perror("cat");

```

```

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

```

```

372         return NULL;
373     }
374 }
375
376 if (out_fd != STDOUT_FILENO) {
377     k_close(out_fd);
378 }
379 return NULL;
380 }
381
382 // handle concatenating one or more files: cat FILE ... [-w/-a OUTPUT_FILE]
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
391 for (i = start; i <= end; i++) {
392     // skip the redirection flags and their arguments
393     if (strcmp(args[i], "-w") == 0 || strcmp(args[i], "-a") == 0) {
394         i++;
395         continue;
396     }
397
398     // open the current input file
399     int in_fd = k_open(args[i], F_READ);
400     if (in_fd < 0) {
401         u_perror("cat");
402         continue;
403     }
404
405     // use lseek to get the size of in_fd
406     off_t in_fd_size = k_lseek(in_fd, 0, SEEK_END);
407     if (in_fd_size == -1) {
408         k_close(in_fd);
409         u_perror("cat");
410         continue;
411     }
412
413     // use lseek to reset position to 0 for reading
414     if (k_lseek(in_fd, 0, SEEK_SET) == -1) {
415         k_close(in_fd);
416         u_perror("cat");
417         continue;
418     }
419
420     // copy file content to output
421     char* buffer = (char*)malloc(block_size);
422     if (buffer == NULL) {
423         P_ERRNO = P_EMALLOC;
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;
435         bytes_read = k_read(in_fd, buffer, bytes_to_read);
436
437         if (bytes_read <= 0) {
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
451     // read error
452     if (bytes_read < 0) {
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()

```

void* chmod (
    void * arg )

```

Changes file permissions.

Modifies the permissions of the specified file.

Usage formats:

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

Parameters

<i>arg</i>	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.

```

770     {
771     char** args = (char**)arg;
772     if (!args || !args[0] || !args[1] || !args[2]) {
773         P_ERRNO = P_EINVAL;
774         return NULL;
775     }
776
777     // Parse permission string
778     const char* perm_str = args[1];
779     if (perm_str[0] != '+' && perm_str[0] != '-') {
780         P_ERRNO = P_EINVAL;
781         return NULL;

```

```

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

<i>arg</i>	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.

```

851     {
852         if (!is_mounted) {
853             P_ERRNO = P_EFS_NOT_MOUNTED;
854             u_perror("cmpctdir");
855             return NULL;
856         }
857         if (compact_directory() != 0) {
858             u_perror("cmpctdir");
859         }
860     }
861     return NULL;
862 }
863 }
```

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

<i>arg</i>	Arguments array (command line arguments)
------------	--

Returns

return 0 on success, -1 on error

Copies files.

Definition at line 640 of file fat_routines.c.

```

640     {
641         char** args = (char**)arg;
642         // check that we have enough arguments
643         if (args[1] == NULL || args[2] == NULL) {
644             P_ERRNO = P_EINVAL;
645             u_perror("cp");
646             return NULL;
647         }
648     }
649 }
```

```

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

<i>arg</i>	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.

```

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");
483             return NULL;
484         }
485     } else {
486         if (k_ls(NULL) == -1) {
487             u_perror("ls");
488             return NULL;
489         }
490     }
491
492     return NULL;
493 }
```

4.2.2.6 mkfs()

```

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

Creates a PennFAT filesystem in the file named `fs_name`.

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

Parameters

<i>fs_name</i>	The name of the file to create the filesystem in.
<i>num_blocks</i>	The number of blocks in the FAT region (1-32).
<i>block_size</i>	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.

```

34
35 // validate arguments
36 if (num_blocks < 1 || num_blocks > 32) {
37     P_ERRNO = P_EINVAL;
38     return -1;
39 }
40 if (blk_size < 0 || blk_size > 4) {
41     P_ERRNO = P_EINVAL;
42     return -1;
43 }
44
45 // determine the file system size
46 int block_sizes[] = {256, 512, 1024, 2048, 4096};
47 int actual_block_size = block_sizes[blk_size];
48 int fat_size = num_blocks * actual_block_size;
49 int fat_entries = fat_size / 2;
50 int num_data_blocks =
51     (num_blocks == 32)
52     ? fat_entries - 2
53     : fat_entries - 1; // note: first entry is reserved for metadata!
54 size_t filesystem_size = fat_size + (actual_block_size * num_data_blocks);
55
56 // create the file for the filesystem
57 int fd = open(fs_name, O_RDWR | O_CREAT | O_TRUNC, 0644);
58 if (fd == -1) {
```

```

59     P_ERRNO = P_EOPEN;
60     return -1;
61 }
62
63 // extend the file to the required size
64 if (ftruncate(fd, filesystem_size) == -1) {
65     P_ERRNO = P_EFUNC;
66     close(fd);
67     return -1;
68 }
69
70 // allocate the FAT
71 uint16_t* temp_fat = (uint16_t*)calloc(fat_entries, sizeof(uint16_t));
72 if (!temp_fat) {
73     P_ERRNO = P_EMALLOC;
74     close(fd);
75     return -1;
76 }
77
78 // initialize FAT entries to their correct values
79 temp_fat[0] = (num_blocks < 8) | blk_size;
80 temp_fat[1] = FAT_EOF;
81 for (int i = 2; i < fat_entries; i++) {
82     temp_fat[i] = FAT_FREE;
83 }
84
85 // 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
93 // initialize the root directory + write to memory
94 uint8_t* root_dir = (uint8_t*)calloc(actual_block_size, 1);
95 if (lseek(fd, fat_size, SEEK_SET) == -1) {
96     P_ERRNO = P_ELSEEK;
97     free(temp_fat);
98     free(root_dir);
99     close(fd);
100    return -1;
101 }
102 if (write(fd, root_dir, actual_block_size) != actual_block_size) {
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;
115 }

```

4.2.2.7 mount()

```

int mount (
    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

<code>fs_name</code>	The name of the filesystem file to mount.
----------------------	---

Returns

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

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

Definition at line 120 of file fat_routines.c.

```

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);
129     if (fs_fd == -1) {
130         P_ERRNO = P_ENOENT;
131         return -1;
132     }
133
134     // read the first two bytes to get size configuration
135     uint16_t config;
136     if (read(fs_fd, &config, sizeof(config)) != sizeof(config)) {
137         P_ERRNO = P_EREAD;
138         close(fs_fd);
139         fs_fd = -1;
140         return -1;
141     }
142
143     // extract FAT region size information
144     num_fat_blocks = (config >> 8) & 0xFF; // MSB
145     int block_size_config = config & 0xFF; // LSB
146     int block_sizes[] = {256, 512, 1024, 2048, 4096};
147     block_size = block_sizes[block_size_config];
148     fat_size = num_fat_blocks * block_size;
149
150     // map the FAT region into memory
151     if (lseek(fs_fd, 0, SEEK_SET) == -1) {
152         P_ERRNO = P_ELSEEK;
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);
162         fs_fd = -1;
163         return -1;
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()

```

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

<i>arg</i>	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) {
566         P_ERRNO = P_EFS_NOT_MOUNTED;
567         u_perror("mv");
568         return NULL;
569     }
570
571     // check if we have both source and destination arguments
572     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         return NULL;
584     }
585
586     // check if source file exists
587     dir_entry_t source_entry;
588     int source_offset = find_file(source, &source_entry);
589     if (source_offset < 0) {
590         u_perror("mv");
591         return NULL;
592     }
593
594     // check if the destination file already exists
595     dir_entry_t dest_entry;
596     int dest_offset = find_file(dest, &dest_entry);
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++) {
602             if (fd_table[i].in_use && strcmp(fd_table[i].filename, dest) == 0) {
603                 P_ERRNO = P_EBUSY;
604                 u_perror("mv");
605                 return NULL;
606             }
607         }
608
609         // if destination file exists, delete it
610         if (mark_entry_as_deleted(&dest_entry, dest_offset) != 0) {
611             u_perror("mv");
612             return NULL;
613         }
614     }
615
616     // rename file
617     strncpy(source_entry.name, dest, sizeof(source_entry.name) - 1);
618     source_entry.name[sizeof(source_entry.name) - 1] = '\0';
619
620     // write the updated entry back to disk
621     if (lseek(fs_fd, source_offset, SEEK_SET) == -1) {
622         P_ERRNO = P_ELSEEK;
623         u_perror("mv");
624         return NULL;
625     }
626
627     if (write(fs_fd, &source_entry, sizeof(source_entry)) !=
628         sizeof(source_entry)) {
629         P_ERRNO = P_EWRITE;
630         u_perror("mv");
631         return NULL;

```

```

632  }
633
634  return NULL;
635 }

```

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

<i>arg</i>	Arguments array (command line arguments)
------------	--

Returns

void pointer (unused)

Definition at line 698 of file fat_routines.c.

```

698  {
699  char** args = (char**)arg;
700
701  // verify that the file system is mounted
702  if (!is_mounted) {
703      P_ERRNO = P_EFS_NOT_MOUNTED;
704      u_perror("rm");
705      return NULL;
706  }
707
708  // check if we have any arguments
709  if (args[1] == NULL) {
710      P_ERRNO = P_EINVAL;
711      u_perror("rm");
712      return NULL;
713  }
714
715  // process each file argument
716  for (int i = 1; args[i] != NULL; i++) {
717      // find the file in the directory
718      dir_entry_t entry;
719      int entry_offset = find_file(args[i], &entry);
720
721      if (entry_offset < 0) {
722          // file doesn't exist
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++) {
730          if (fd_table[j].in_use && strcmp(fd_table[j].filename, args[i]) == 0) {
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) {
739          P_ERRNO = P_ELSEEK;
740          u_perror("rm");
741          continue;
742      }

```

```

743
744     char deleted = 1; // mark as deleted
745     if (write(fs_fd, &deleted, sizeof(deleted)) != sizeof(deleted)) {
746         P_ERRNO = P_EWRITE;
747         u_perror("rm");
748         continue;
749     }
750
751     // free the FAT chain for this file
752     uint16_t block = entry.firstBlock;
753     while (block != FAT_FREE && block != FAT_EOF) {
754         uint16_t next_block = fat[block];
755         fat[block] = FAT_FREE;
756         block = next_block;
757     }
758 }
759
760 return NULL;
761 }

```

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

<i>arg</i>	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.

```

501     {
502     char** args = (char**)arg;
503
504     // verify that the file system is mounted
505     if (!is_mounted) {
506         P_ERRNO = P_EFS_NOT_MOUNTED;
507         u_perror("touch");
508         return NULL;
509     }
510
511     // check if we have any arguments
512     if (args[1] == NULL) {
513         P_ERRNO = P_EINVAL;
514         u_perror("touch");
515         return NULL;
516     }
517
518     // process each file argument
519     for (int i = 1; args[i] != NULL; i++) {
520         dir_entry_t entry;
521         int entry_offset = find_file(args[i], &entry);
522
523         // file exists
524         if (entry_offset >= 0) {
525             entry.mtime = time(NULL);

```



```

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

```

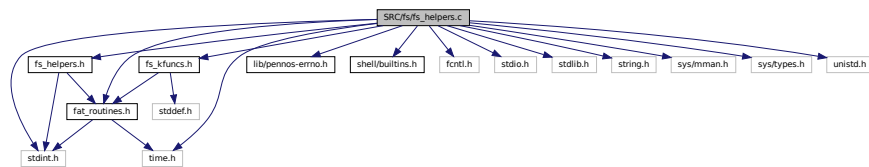
174 {
175     // first check that a file system is actually mounted
176     if (!is_mounted) {
177         P_ERRNO = P_EFS_NOT_MOUNTED;
178         return -1;
179     }
180
181     // unmap the FAT
182     if (fat != NULL) {
183         if (munmap(fat, fat_size) == -1) {
184             P_ERRNO = P_EMAP;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

Include dependency graph for fs_helpers.c:



Functions

- void `init_fd_table` (`fd_entry_t *fd_table`)
Initializes the global kernel-level file descriptor table.
- int `get_free_fd` (`fd_entry_t *fd_table`)
Gets a free file descriptor.
- int `increment_fd_ref_count` (int fd)
Increments the reference count of a file descriptor.
- int `decrement_fd_ref_count` (int fd)
Decrements the reference count of a file descriptor.
- int `has_executable_permission` (int fd)
Checks if a file has executable permissions.
- uint16_t `allocate_block` ()
Allocates a block.
- int `find_file` (const char *filename, `dir_entry_t *entry`)
Searches for a file in the root directory.
- int `add_file_entry` (const char *filename, uint32_t size, uint16_t first_block, uint8_t type, uint8_t perm)
Adds a file to the root directory.
- int `mark_entry_as_deleted` (`dir_entry_t *entry`, int absolute_offset)
Marks a file entry as deleted and frees its blocks.
- int `copy_host_to_pennfat` (const char *host_filename, const char *pennfat_filename)
Copies data from host OS file to the PennFAT file.
- int `copy_pennfat_to_host` (const char *pennfat_filename, const char *host_filename)
Copies data from PennFAT file to host OS file.
- int `copy_source_to_dest` (const char *source_filename, const char *dest_filename)
Copies data from source file to destination file.
- int `compact_directory` ()
Compacts a directory.

Variables

- int `fs_fd` = -1
- int `block_size` = 0
- int `num_fat_blocks` = 0
- int `fat_size` = 0
- uint16_t * `fat` = NULL
- bool `is_mounted` = false
- int `MAX_FDS` = 100
- fd_entry_t `fd_table` [100]

4.3.1 Function Documentation

4.3.1.1 add_file_entry()

```
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.

Adds a new file entry to the root directory.

Definition at line 267 of file fs_helpers.c.

```
271                                     {
272     if (!is_mounted) {
273         P_ERRNO = P_EFS_NOT_MOUNTED;
274         return -1;
275     }
276
277     // check if file already exists
278     dir_entry_t existing;
279     if (find_file(filename, &existing) >= 0) {
280         P_ERRNO = P_EEXIST;
281         return -1;
282     }
283
284     // start with root directory block (block 1)
285     uint16_t current_block = 1;
286     int offset = 0;
287     dir_entry_t dir_entry;
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) ==
292             -1) {
293             P_ERRNO = P_ELSEEK;
294             return -1;
295         }
296
297         // reset offset for new block
298         offset = 0;
299
300         // search current block for free slot
301         while (offset < block_size) {
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
308             if (dir_entry.name[0] == 0 || dir_entry.name[0] == 1) {
```

```

309         // initialize the new entry
310         memset(&dir_entry, 0, sizeof(dir_entry));
311         strncpy(dir_entry.name, filename, 31);
312         dir_entry.size = size;
313         dir_entry.firstBlock = first_block;
314         dir_entry.type = type;
315         dir_entry.perm = perm;
316         dir_entry.mtime = time(NULL);
317
318         // write the entry
319         if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + offset,
320                 SEEK_SET) == -1) {
321             P_ERRNO = P_ELSEEK;
322             return -1;
323         }
324         if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
325             P_ERRNO = P_EWRITE;
326             return -1;
327         }
328
329         return offset;
330     }
331
332     offset += sizeof(dir_entry);
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();
343 if (new_block == 0) {
344     P_ERRNO = P_EFULL;
345     return -1;
346 }
347
348 // chain the new block
349 fat[current_block] = new_block;
350 fat[new_block] = FAT_EOF;
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;
362     free(zero_block);
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;
377 dir_entry.firstBlock = first_block;
378 dir_entry.type = type;
379 dir_entry.perm = perm;
380 dir_entry.mtime = time(NULL);
381
382 // write the new entry at the start of the new block in the file system
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 }

```

4.3.1.2 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.

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

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

```

725
726     if (read(fs_fd, dir_buffer, block_size) != block_size) {
727         P_ERRNO = P_EREAD;
728         free(dir_buffer);
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)) {
734         dir_entry_t* entry = (dir_entry_t*)(dir_buffer + offset);
735
736         // check if we've reached the end of directory
737         if (entry->name[0] == 0) {
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) {
766     P_ERRNO = P_EMALLOC;
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) {
785         P_ERRNO = P_EREAD;
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)) {
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
800         // skip deleted entries
801         if (entry->name[0] == 1) {
802             continue;
803         }
804
805         // copy valid entry to our array
806         memcpy(&all_entries[valid_entry_idx++], entry, sizeof(dir_entry_t));
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;
819 int entries_per_block = block_size / sizeof(dir_entry_t);
820 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
824 uint16_t next_block = fat[current_block];
825 if (blocks_needed == 1) {
826     // only need one block, free all others
827     while (next_block != FAT_EOF) {
828         uint16_t temp = fat[next_block];
829         fat[next_block] = FAT_FREE;
830         next_block = temp;
831     }
832     fat[current_block] = FAT_EOF;
833 } else {
834     // navigate through needed blocks
835     int block_count = 1;
836     uint16_t prev_block = current_block;
837
838     while (block_count < blocks_needed) {
839         if (next_block == FAT_EOF) {
840             // need to allocate a new block
841             uint16_t new_block = allocate_block();
842             if (new_block == 0) {
843                 P_ERRNO = P_EFULL;
844                 free(dir_buffer);
845                 free(all_entries);
846                 return -1;
847             }
848             fat[prev_block] = new_block;
849             next_block = new_block;
850         }
851
852         prev_block = next_block;
853         next_block = fat[next_block];
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];
861         fat[next_block] = FAT_FREE;
862         next_block = temp;
863     }
864 }
865
866 // write the valid entries back to the directory blocks
867 current_block = 1;
868 int entries_written = 0;
869
870 while (entries_written < valid_entry_idx) {
871     if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
872         -1) {
873         P_ERRNO = P_ELSEEK;
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
882     int entries_in_this_block = 0;
883     while (entries_written < valid_entry_idx &&
884         entries_in_this_block < entries_per_block) {
885         memcpy(dir_buffer + (entries_in_this_block * sizeof(dir_entry_t)),
886             &all_entries[entries_written], sizeof(dir_entry_t));
887         entries_written++;
888         entries_in_this_block++;
889     }
890
891     // write the buffer to the file system
892     if (write(fs_fd, dir_buffer, block_size) != block_size) {
893         P_ERRNO = P_EINVAL;
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 }

```

4.3.1.4 copy_host_to_pennfat()

```

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

```

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.

```

439                                     {
440     if (!is_mounted) {
441         P_ERRNO = P_EFS_NOT_MOUNTED;
442         return -1;
443     }
444
445     // open the host file
446     int host_fd = open(host_filename, O_RDONLY);
447     if (host_fd == -1) {
448         P_ERRNO = P_EOPEN;
449         return -1;
450     }
451
452     // determine file size by seeking to the end and getting position
453     off_t host_file_size_in_bytes = lseek(host_fd, 0, SEEK_END);
454     if (host_file_size_in_bytes == -1) {
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
468     int pennfat_fd = k_open(pennfat_filename, F_WRITE);
469     if (pennfat_fd < 0) {
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);
476     if (!buffer) {
477         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
487     while (bytes_remaining > 0) {
488         // ensure bytes to read never exceeds the block size
489         ssize_t bytes_to_read =
490             bytes_remaining < block_size ? bytes_remaining : block_size;
491         bytes_read = read(host_fd, buffer, bytes_to_read);
492     }

```



```

493     if (bytes_read <= 0) {
494         break;
495     }
496
497     // write to pennfat_fd using k_write
498     if (k_write(pennfat_fd, (const char*)buffer, bytes_read) != bytes_read) {
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
509 if (bytes_read < 0) {
510     P_ERRNO = P_EREAD;
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()

```

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

```

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
535     int pennfat_fd = k_open(pennfat_filename, F_READ);
536     if (pennfat_fd < 0) {
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);
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) {
556         P_ERRNO = P_EOPEN;
557         k_close(pennfat_fd);
558         return -1;
559     }
560
561     // allocate buffer for data transfer

```

```

562 char* buffer = (char*)malloc(block_size);
563 if (!buffer) {
564     P_ERRNO = P_EMALLOC;
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         bytes_remaining < block_size ? bytes_remaining : block_size;
578     bytes_read = k_read(pennfat_fd, buffer, bytes_to_read);
579
580     if (bytes_read <= 0) {
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
596 if (bytes_read < 0) {
597     P_ERRNO = P_EREAD;
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 }

```

4.3.1.6 copy_source_to_dest()

```

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

```

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) {
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);
623     if (source_fd < 0) {
624         return -1;
625     }
626
627     // get the source file size
628     off_t source_file_size_in_bytes = k_lseek(source_fd, 0, SEEK_END);
629     if (source_file_size_in_bytes == -1) {
630         k_close(source_fd);

```

```

631     return -1;
632 }
633
634 // move to the beginning of the source file for reading
635 if (k_lseek(source_fd, 0, SEEK_SET) < 0) {
636     k_close(source_fd);
637     return -1;
638 }
639
640 // open the destination file
641 int dest_fd = k_open(dest_filename, F_WRITE);
642 if (dest_fd < 0) {
643     k_close(source_fd);
644     return -1;
645 }
646
647 // read from source to destination
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) {
660     // make sure the bytes to read doesn't exceed block size
661     ssize_t bytes_to_read =
662         bytes_remaining < block_size ? bytes_remaining : block_size;
663     bytes_read = k_read(source_fd, buffer, bytes_to_read);
664
665     if (bytes_read <= 0) {
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) {
679     free(buffer);
680     k_close(source_fd);
681     k_close(dest_fd);
682     return -1;
683 }
684
685 // otherwise, cleanup and return success
686 free(buffer);
687 k_close(source_fd);
688 k_close(dest_fd);
689 return 0;
690 }

```

4.3.1.7 decrement_fd_ref_count()

```

int decrement_fd_ref_count (
    int fd )

```

Decrements the reference count of a file descriptor.

If reference count reaches 0, flush field values.

Definition at line 107 of file fs_helpers.c.

```

107 {
108     if (fd < 0 || fd >= MAX_FDS) {
109         P_ERRNO = P_EBADF;
110         return -1;
111     }

```

```

112
113 if (!fd_table[fd].in_use) {
114     P_ERRNO = P_EBADF;
115     return -1;
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));
122     fd_table[fd].size = 0;
123     fd_table[fd].first_block = 0;
124     fd_table[fd].position = 0;
125     fd_table[fd].mode = 0;
126 }
127 return fd_table[fd].ref_count;
128 }

```

4.3.1.8 find_file()

```

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

```

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
197 // Start with root directory block (block 1)
198 uint16_t current_block = 1;
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
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) {
219         if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
220             P_ERRNO = P_EREOF;
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
229         // check if this is a deleted entry
230         if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
231             offset_in_block += sizeof(dir_entry);
232             absolute_offset += sizeof(dir_entry);
233             continue;
234         }
235
236         // check if we found the file
237         if (strcmp(dir_entry.name, filename) == 0) {

```

```

238     if (entry) {
239         memcpy(entry, &dir_entry, sizeof(dir_entry));
240     }
241     return absolute_offset; // return the absolute file offset
242 }
243
244 offset_in_block += sizeof(dir_entry);
245 absolute_offset += sizeof(dir_entry);
246 }
247
248 // if we've reached the end of the current block, check if there's a next
249 // block
250 if (fat[current_block] != FAT_EOF) {
251     current_block = fat[current_block];
252     continue;
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 }

```

4.3.1.9 get_free_fd()

```

int get_free_fd (
    fd_entry_t * fd_table )

```

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 }

```

4.3.1.10 has_executable_permission()

```

int has_executable_permission (
    int fd )

```

Checks if a file has executable permissions.

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

Definition at line 133 of file fs_helpers.c.

```

133 {
134     // check if fs is mounted
135     if (!is_mounted) {
136         P_ERRNO = P_EFS_NOT_MOUNTED;
137         return -1;
138     }
139
140     // validate fd argument
141     if (fd < 0 || fd >= MAX_FDS) {
142         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) {
150     return -1;
151 }
152
153 // if it exists, get its permission
154 if (entry.perm & PERM_EXEC) {
155     return 1;
156 }
157
158 return 0;
159 }

```

4.3.1.11 increment_fd_ref_count()

```

int increment_fd_ref_count (
    int fd )

```

Increments the reference count of a file descriptor.

Parameters

<i>fd</i>	file descriptor to increment
-----------	------------------------------

Returns

new reference count, or -1 on error

Definition at line 89 of file fs_helpers.c.

```

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

```

4.3.1.12 init_fd_table()

```

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

```

4.3.1.13 mark_entry_as_deleted()

```

int mark_entry_as_deleted (
    dir_entry_t * entry,
    int absolute_offset )

```

Marks a file entry as deleted and frees its blocks.

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

Definition at line 399 of file fs_helpers.c.

```

399 {
400     if (!is_mounted || entry == NULL || absolute_offset < 0) {
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) {
408         uint16_t next_block = fat[current_block];
409         fat[current_block] = FAT_FREE;
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;
416     if (lseek(fs_fd, absolute_offset, SEEK_SET) == -1) {
417         P_ERRNO = P_ELSEEK;
418         return -1;
419     }
420     if (write(fs_fd, &deleted_entry, sizeof(deleted_entry)) !=
421         sizeof(deleted_entry)) {
422         P_ERRNO = P_EINVAL;
423         return -1;
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
```

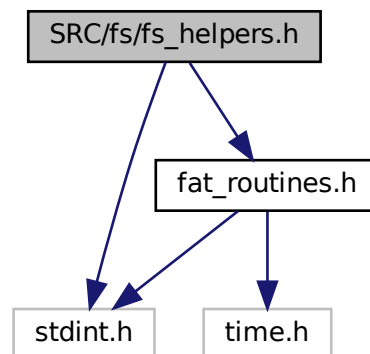
Definition at line 29 of file fs_helpers.c.

4.4 SRC/fs/fs_helpers.h File Reference

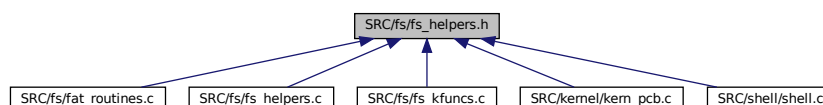
```
#include <stdint.h>
```

```
#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()`

```
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.

Parameters

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

```

271     {
272     if (!is_mounted) {
273         P_ERRNO = P_EFS_NOT_MOUNTED;
274         return -1;
275     }
276
277     // check if file already exists
278     dir_entry_t existing;
279     if (find_file(filename, &existing) >= 0) {
280         P_ERRNO = P_EEXIST;
281         return -1;
282     }
283
284     // start with root directory block (block 1)
285     uint16_t current_block = 1;
286     int offset = 0;
287     dir_entry_t dir_entry;
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) ==
292             -1) {
293             P_ERRNO = P_ELSEEK;
294             return -1;
295         }
296
297         // reset offset for new block
298         offset = 0;
299
300         // search current block for free slot
301         while (offset < block_size) {
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
308             if (dir_entry.name[0] == 0 || dir_entry.name[0] == 1) {
309                 // initialize the new entry
310                 memset(&dir_entry, 0, sizeof(dir_entry));
311                 strncpy(dir_entry.name, filename, 31);
312                 dir_entry.size = size;
313                 dir_entry.firstBlock = first_block;
314                 dir_entry.type = type;
315                 dir_entry.perm = perm;
316                 dir_entry.mtime = time(NULL);
317
318                 // write the entry
319                 if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + offset,
320                     SEEK_SET) == -1) {
321                     P_ERRNO = P_ELSEEK;
322                     return -1;
323                 }
324                 if (write(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
325                     P_ERRNO = P_EWRITE;
326                     return -1;
327                 }
328
329                 return offset;
330             }
331         }

```

```

332     offset += sizeof(dir_entry);
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();
343 if (new_block == 0) {
344     P_ERRNO = P_EFULL;
345     return -1;
346 }
347
348 // chain the new block
349 fat[current_block] = new_block;
350 fat[new_block] = FAT_EOF;
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;
362     free(zero_block);
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;
377 dir_entry.firstBlock = first_block;
378 dir_entry.type = type;
379 dir_entry.perm = perm;
380 dir_entry.mtime = time(NULL);
381
382 // write the new entry at the start of the new block in the file system
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++) {
168     if (fat[i] == FAT_FREE) {
169     fat[i] = FAT_EOF;
170     return i;
171     }
172     }
173
174     if (compact_directory() == 0) {
175     for (int i = 2; i < fat_size / 2; i++) {
176     if (fat[i] == FAT_FREE) {
177     fat[i] = FAT_EOF;
178     return i;
179     }
180     }
181     }
182
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
705     // buffer for temp storage of a block
706     uint8_t* dir_buffer = malloc(block_size);
707     if (!dir_buffer) {
708     P_ERRNO = P_EMALLOC;
709     return -1;
710     }
711
712     // start at root directory
713     uint16_t current_block = 1;
714     int dir_entries_count = 0;
715     int deleted_entries_count = 0;
716
717     // 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) ==
720     -1) {
721     P_ERRNO = P_ELSEEK;
722     free(dir_buffer);
723     return -1;
724     }
725
726     if (read(fs_fd, dir_buffer, block_size) != block_size) {
```

```

727     P_ERRNO = P_EREAD;
728     free(dir_buffer);
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)) {
734     dir_entry_t* entry = (dir_entry_t*)(dir_buffer + offset);
735
736     // check if we've reached the end of directory
737     if (entry->name[0] == 0) {
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) {
766     P_ERRNO = P_EMALLOC;
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) {
785         P_ERRNO = P_EREAD;
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)) {
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
800         // skip deleted entries
801         if (entry->name[0] == 1) {
802             continue;
803         }
804
805         // copy valid entry to our array
806         memcpy(&all_entries[valid_entry_idx++], entry, sizeof(dir_entry_t));
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;
819 int entries_per_block = block_size / sizeof(dir_entry_t);
820 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
824 uint16_t next_block = fat[current_block];
825 if (blocks_needed == 1) {
826     // only need one block, free all others
827     while (next_block != FAT_EOF) {
828         uint16_t temp = fat[next_block];
829         fat[next_block] = FAT_FREE;
830         next_block = temp;
831     }
832     fat[current_block] = FAT_EOF;
833 } else {
834     // navigate through needed blocks
835     int block_count = 1;
836     uint16_t prev_block = current_block;
837
838     while (block_count < blocks_needed) {
839         if (next_block == FAT_EOF) {
840             // need to allocate a new block
841             uint16_t new_block = allocate_block();
842             if (new_block == 0) {
843                 P_ERRNO = P_EFULL;
844                 free(dir_buffer);
845                 free(all_entries);
846                 return -1;
847             }
848             fat[prev_block] = new_block;
849             next_block = new_block;
850         }
851
852         prev_block = next_block;
853         next_block = fat[next_block];
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];
861         fat[next_block] = FAT_FREE;
862         next_block = temp;
863     }
864 }
865
866 // write the valid entries back to the directory blocks
867 current_block = 1;
868 int entries_written = 0;
869
870 while (entries_written < valid_entry_idx) {
871     if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
872         -1) {
873         P_ERRNO = P_ELSEEK;
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
882     int entries_in_this_block = 0;
883     while (entries_written < valid_entry_idx &&
884         entries_in_this_block < entries_per_block) {
885         memcpy(dir_buffer + (entries_in_this_block * sizeof(dir_entry_t)),
886             &all_entries[entries_written], sizeof(dir_entry_t));
887         entries_written++;
888         entries_in_this_block++;
889     }
890
891     // write the buffer to the file system
892     if (write(fs_fd, dir_buffer, block_size) != block_size) {
893         P_ERRNO = P_EINVAL;
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 }

```

4.4.1.4 copy_host_to_pennfat()

```

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

```

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

Parameters

<i>host_filename</i>	path to the file on the host OS
<i>pennfat_filename</i>	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.

```

439                                     {
440     if (!is_mounted) {
441         P_ERRNO = P_EFS_NOT_MOUNTED;
442         return -1;
443     }
444
445     // open the host file
446     int host_fd = open(host_filename, O_RDONLY);
447     if (host_fd == -1) {
448         P_ERRNO = P_EOPEN;
449         return -1;
450     }
451
452     // determine file size by seeking to the end and getting position
453     off_t host_file_size_in_bytes = lseek(host_fd, 0, SEEK_END);
454     if (host_file_size_in_bytes == -1) {
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
468     int pennfat_fd = k_open(pennfat_filename, F_WRITE);
469     if (pennfat_fd < 0) {
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);
476     if (!buffer) {

```



```

477     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
487 while (bytes_remaining > 0) {
488     // ensure bytes to read never exceeds the block size
489     ssize_t bytes_to_read =
490         bytes_remaining < block_size ? bytes_remaining : block_size;
491     bytes_read = read(host_fd, buffer, bytes_to_read);
492
493     if (bytes_read <= 0) {
494         break;
495     }
496
497     // write to pennfat_fd using k_write
498     if (k_write(pennfat_fd, (const char*)buffer, bytes_read) != bytes_read) {
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
509 if (bytes_read < 0) {
510     P_ERRNO = P_EREAD;
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 }

```

4.4.1.5 copy_pennfat_to_host()

```

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

```

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

Parameters

<i>pennfat_filename</i>	name of the file in PennFAT
<i>host_filename</i>	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;
531         return -1;
532     }
533
534     // open the PennFAT file
535     int pennfat_fd = k_open(pennfat_filename, F_READ);
536     if (pennfat_fd < 0) {
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);
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) {
556         P_ERRNO = P_EOPEN;
557         k_close(pennfat_fd);
558         return -1;
559     }
560
561     // allocate buffer for data transfer
562     char* buffer = (char*)malloc(block_size);
563     if (!buffer) {
564         P_ERRNO = P_EMALLOC;
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             bytes_remaining < block_size ? bytes_remaining : block_size;
578         bytes_read = k_read(pennfat_fd, buffer, bytes_to_read);
579
580         if (bytes_read <= 0) {
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
596     if (bytes_read < 0) {
597         P_ERRNO = P_EREAD;
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()

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

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

Parameters

<i>source_filename</i>	name of the source filename
<i>dest_filename</i>	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);
623     if (source_fd < 0) {
624         return -1;
625     }
626
627     // get the source file size
628     off_t source_file_size_in_bytes = k_lseek(source_fd, 0, SEEK_END);
629     if (source_file_size_in_bytes == -1) {
630         k_close(source_fd);
631         return -1;
632     }
633
634     // move to the beginning of the source file for reading
635     if (k_lseek(source_fd, 0, SEEK_SET) < 0) {
636         k_close(source_fd);
637         return -1;
638     }
639
640     // open the destination file
641     int dest_fd = k_open(dest_filename, F_WRITE);
642     if (dest_fd < 0) {
643         k_close(source_fd);
644         return -1;
645     }
646
647     // read from source to destination
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) {
660         // make sure the bytes to read doesn't exceed block size
661         ssize_t bytes_to_read =
662             bytes_remaining < block_size ? bytes_remaining : block_size;
663         bytes_read = k_read(source_fd, buffer, bytes_to_read);
664
665         if (bytes_read <= 0) {
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) {
679     free(buffer);
680     k_close(source_fd);
681     k_close(dest_fd);
682     return -1;
683 }
684
685 // otherwise, cleanup and return success
686 free(buffer);
687 k_close(source_fd);
688 k_close(dest_fd);
689 return 0;
690 }

```

4.4.1.7 decrement_fd_ref_count()

```

int decrement_fd_ref_count (
    int fd )

```

Decrements the reference count of a file descriptor.

Parameters

<i>fd</i>	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
113     if (!fd_table[fd].in_use) {
114         P_ERRNO = P_EBADF;
115         return -1;
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));
122         fd_table[fd].size = 0;
123         fd_table[fd].first_block = 0;
124         fd_table[fd].position = 0;
125         fd_table[fd].mode = 0;
126     }
127     return fd_table[fd].ref_count;
128 }

```

4.4.1.8 find_file()

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

Searches for a file in the root directory.

Parameters

<i>filename</i>	name of the file to find
<i>entry</i>	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)
198     uint16_t current_block = 1;
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
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) {
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
229             // check if this is a deleted entry
230             if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
231                 offset_in_block += sizeof(dir_entry);
232                 absolute_offset += sizeof(dir_entry);
233                 continue;
234             }
235
236             // check if we found the file
237             if (strcmp(dir_entry.name, filename) == 0) {
238                 if (entry) {
239                     memcpy(entry, &dir_entry, sizeof(dir_entry));
240                 }
241                 return absolute_offset; // return the absolute file offset
242             }
243
244             offset_in_block += sizeof(dir_entry);
```

```

245     absolute_offset += sizeof(dir_entry);
246 }
247
248 // if we've reached the end of the current block, check if there's a next
249 // block
250 if (fat[current_block] != FAT_EOF) {
251     current_block = fat[current_block];
252     continue;
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()

```

int get_free_fd (
    fd_entry_t * fd_table )

```

Finds the first available file descriptor in the table.

Parameters

<i>fd_table</i>	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 }

```

4.4.1.10 has_executable_permission()

```

int has_executable_permission (
    int fd )

```

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

Parameters

<i>fd</i>	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) {
136         P_ERRNO = P_EFS_NOT_MOUNTED;
137         return -1;
138     }
139
140     // validate fd argument
141     if (fd < 0 || fd >= MAX_FDS) {
142         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) {
150         return -1;
151     }
152
153     // if it exists, get its permission
154     if (entry.perm & PERM_EXEC) {
155         return 1;
156     }
157
158     return 0;
159 }
```

4.4.1.11 increment_fd_ref_count()

```

int increment_fd_ref_count (
    int fd )
```

Increments the reference count of a file descriptor.

Parameters

<i>fd</i>	file descriptor to increment
-----------	------------------------------

Returns

new reference count, or -1 on error

Definition at line 89 of file fs_helpers.c.

```

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

4.4.1.12 init_fd_table()

```
void init_fd_table (
    fd_entry_t * fd_table )
```

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

Parameters

<i>fd_table</i>	pointer to the file descriptor table to initialize
-----------------	--

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

Definition at line 43 of file fs_helpers.c.

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

4.4.1.13 mark_entry_as_deleted()

```
int mark_entry_as_deleted (
    dir_entry_t * entry,
    int absolute_offset )
```

Marks a file entry as deleted and frees its blocks 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

<i>entry</i>	the entry struct of the file to mark as deleted.
<i>offset</i>	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 {
400     if (!is_mounted || entry == NULL || absolute_offset < 0) {
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) {
408         uint16_t next_block = fat[current_block];
409         fat[current_block] = FAT_FREE;
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;
416     if (lseek(fs_fd, absolute_offset, SEEK_SET) == -1) {
417         P_ERRNO = P_ELSEEK;
418         return -1;
419     }
420     if (write(fs_fd, &deleted_entry, sizeof(deleted_entry)) !=
421         sizeof(deleted_entry)) {
422         P_ERRNO = P_EINVAL;
423         return -1;
424     }
425
426     // mark the passed entry as deleted
427     entry->name[0] = 1;
428     return 0;
429 }

```

4.4.2 Variable Documentation

4.4.2.1 block_size

int block_size [extern]

Definition at line 28 of file fs_helpers.c.

4.4.2.2 fat

uint16_t* fat [extern]

Definition at line 31 of file fs_helpers.c.

4.4.2.3 fat_size

```
int fat_size [extern]
```

Definition at line 30 of file fs_helpers.c.

4.4.2.4 fd_table

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

Definition at line 34 of file fs_helpers.c.

4.4.2.5 fs_fd

```
int fs_fd [extern]
```

Definition at line 27 of file fs_helpers.c.

4.4.2.6 is_mounted

```
bool is_mounted [extern]
```

Definition at line 32 of file fs_helpers.c.

4.4.2.7 MAX_FDS

```
int MAX_FDS [extern]
```

Definition at line 33 of file fs_helpers.c.

4.4.2.8 num_fat_blocks

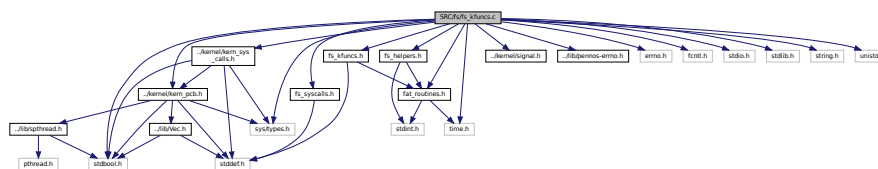
```
int num_fat_blocks [extern]
```

Definition at line 29 of file fs_helpers.c.

4.5 SRC/fs/fs_kfuncs.c File Reference

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

Include dependency graph for fs_kfuncs.c:



Functions

- `int k_open` (const char *fname, int mode)
Kernel-level call to open a file.
- `int k_read` (int fd, char *buf, int n)
Kernel-level call to read a file.
- `int k_write` (int fd, const char *str, int n)
Kernel-level call to write to a file.
- `int k_close` (int fd)
Kernel-level call to close a file.
- `int k_unlink` (const char *fname)
Kernel-level call to remove a file.
- `int k_lseek` (int fd, int offset, int whence)
Kernel-level call to re-position a file offset.
- `int k_ls` (const char *filename)
Kernel-level call to list files.

Variables

- `pcb_t * current_running_pcb`
- `pid_t current_fg_pid`

4.5.1 Function Documentation

4.5.1.1 k_close()

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

Kernel-level call to close a file.

Closes an open file.

Definition at line 526 of file fs_kfuncs.c.

```
526     {
527     // validate the file descriptor
528     if (fd < 0 || fd >= MAX_FDS) {
529         P_ERRNO = P_EBADF;
530         return -1;
531     }
532
533     // ensure any pending changes are written to disk
534     // update the directory entry with the current file size
535     dir_entry_t entry;
536     int file_offset = find_file(fd_table[fd].filename, &entry);
537
538     if (file_offset >= 0) {
539         // update file size if it changed
540         if (entry.size != fd_table[fd].size) {
541             entry.size = fd_table[fd].size;
542             entry.mtime = time(NULL);
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)) {
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()

```
int k_ls (
    const char * filename )
```

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;
```

```

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
681          if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
682              -1) {
683              P_ERRNO = P_ELSEEK;
684              return -1;
685          }
686
687          offset = 0;
688
689          // search current block
690          while (offset < block_size) {
691              if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
692                  P_ERRNO = P_EREAD;
693                  return -1;
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
702              if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
703                  offset += sizeof(dir_entry);
704                  continue;
705              }
706
707              // format permission string
708              char perm_str[4] = "---";
709              if (dir_entry.perm & PERM_READ)
710                  perm_str[0] = 'r';
711              if (dir_entry.perm & PERM_WRITE)
712                  perm_str[1] = 'w';
713              if (dir_entry.perm & PERM_EXEC)
714                  perm_str[2] = 'x';
715
716              // format time
717              struct tm* tm_info = localtime(&dir_entry.mtime);
718              char time_str[50];
719              strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);
720
721              // print entry details
722              char buffer[128];
723              int len;
724              if (dir_entry.firstBlock == 0) {
725                  len = snprintf(buffer, sizeof(buffer), "  -%-6d %s %s\n",
726                                perm_str, dir_entry.size, time_str, dir_entry.name);
727              } else {
728                  len = snprintf(buffer, sizeof(buffer), "%2d -%-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;
735                  return -1;
736              }
737
738              if (k_write(STDOUT_FILENO, buffer, len) != len) {
739                  P_ERRNO = P_EWRITE;
740                  return -1;
741              }
742
743              offset += sizeof(dir_entry);
744          }
745
746          // 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
752          // no more blocks to search
753          break;
754      }
755  } else {
756      // find and display specific file
757      int file_offset = find_file(filename, &dir_entry);
758      if (file_offset < 0) {
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
769     if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
770         P_ERRNO = P_ENOENT;
771         return -1;
772     }
773
774     // format permission string
775     char perm_str[4] = "---";
776     if (dir_entry.perm & PERM_READ)
777         perm_str[0] = 'r';
778     if (dir_entry.perm & PERM_WRITE)
779         perm_str[1] = 'w';
780     if (dir_entry.perm & PERM_EXEC)
781         perm_str[2] = 'x';
782
783     // format time
784     struct tm* tm_info = localtime(&dir_entry.mtime);
785     char time_str[50];
786     strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);
787
788     // print entry details
789     char buffer[128];
790     int len;
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 {
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;
802         return -1;
803     }
804
805     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()

```

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

```

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

Repositions the file offset of an open file.

Definition at line 620 of file fs_kfuncs.c.

```

620     {
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
628         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:
638         new_position = offset;
639         break;
640     case SEEK_CUR:
641         new_position = fd_table[fd].position + offset;
642         break;
643     case SEEK_END:
644         new_position = fd_table[fd].size + offset;
645         break;
646     default:
647         P_ERRNO = P_EINVAL;
648         return -1;
649 }
650
651 // check if new position is valid
652 if (new_position < 0) {
653     P_ERRNO = P_EINVAL;
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()

```

int k_open (
    const char * fname,
    int mode )

```

Kernel-level call to open a file.

Opens a file with the specified mode.

Definition at line 31 of file fs_kfuncs.c.

```

31 {
32     // validate arguments
33     if (fname == NULL || *fname == '\0') {
34         P_ERRNO = P_EINVAL;
35         return -1;
36     }
37     if ((mode & (F_READ | F_WRITE | F_APPEND)) == 0) {
38         P_ERRNO = P_EINVAL;
39         return -1;
40     }
41
42     // check if the file system is mounted
43     if (!is_mounted) {
44         P_ERRNO = P_EFS_NOT_MOUNTED;
45         return -1;
46     }
47
48     // get a free file descriptor
49     int fd = get_free_fd(fd_table);
50     if (fd < 0) {
51         P_ERRNO = P_EFULL; // no free file descriptors
52         return -1;
53     }
54
55     // check if the file exists
56     dir_entry_t entry;
57     int file_offset = find_file(fname, &entry);
58
59     // file exists
60     if (file_offset >= 0) {
61         // check if the file is already open in write mode by another descriptor
62         if ((mode & (F_WRITE | F_APPEND)) != 0) {

```

```

63     for (int i = 0; i < MAX_FDS; i++) {
64         if (i != fd && fd_table[i].in_use &&
65             strcmp(fd_table[i].filename, fname) == 0 &&
66             (fd_table[i].mode & (F_WRITE | F_APPEND)) != 0) {
67             P_ERRNO = P_EBUSY; // file is already open for writing
68             return -1;
69         }
70     }
71 }
72
73 // 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] = '\0';
78 fd_table[fd].size = entry.size;
79 fd_table[fd].first_block = entry.firstBlock;
80 fd_table[fd].mode = mode;
81
82 // set the initial position
83 if (mode & F_APPEND) {
84     fd_table[fd].position = entry.size;
85 } else {
86     fd_table[fd].position = 0;
87 }
88
89 // if mode includes F_WRITE and not 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;
93     uint16_t next_block;
94
95     if (block != 0 && block != FAT_EOF) {
96         next_block = fat[block];
97         fat[block] = FAT_EOF; // terminate the chain at the first block
98         block = next_block;
99
100        // free the rest of the chain
101        while (block != 0 && block != FAT_EOF) {
102            next_block = fat[block];
103            fat[block] = FAT_FREE;
104            block = next_block;
105        }
106    }
107
108    // update file size to 0
109    fd_table[fd].size = 0;
110    entry.size = 0;
111    entry.mtime = time(NULL);
112
113    // update the file system with the truncated file
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;
120        return -1;
121    }
122 }
123 } else {
124     // file doesn't exist
125
126     // we can only create it if we are reading the file
127     if (!(mode & F_WRITE)) {
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) {
135         P_ERRNO = P_EFULL;
136         return -1;
137     }
138
139     // create a new file entry
140     if (add_file_entry(fname, 0, first_block, TYPE_REGULAR, PERM_READ_WRITE) ==
141         -1) {
142         // error code already set by add_file_entry
143         fat[first_block] = FAT_FREE;
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] = '\\0';
152     fd_table[fd].size = 0;
153     fd_table[fd].first_block = first_block;
154     fd_table[fd].position = 0;
155     fd_table[fd].mode = mode;
156 }
157
158 return fd;
159 }

```

4.5.1.5 k_read()

```

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

```

Kernel-level call to read a file.

Reads data from an open file.

Definition at line 164 of file fs_kfuncs.c.

```

164     {
165     // 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) {
168             s_kill(current_running_pcb->pid, P_SIGSTOP);
169         }
170     }
171
172     // handle standard input
173     if (fd == STDIN_FILENO) {
174         return read(STDIN_FILENO, buf, n);
175     }
176
177     // validate inputs
178     if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
179         P_ERRNO = P_EBADF;
180         return -1;
181     }
182     if (buf == NULL || n < 0) {
183         P_ERRNO = P_EINVAL;
184         return -1;
185     }
186     if (n == 0) {
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
195     // determine how many bytes we can actually read
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     // find the block containing the current position
202     uint16_t current_block = fd_table[fd].first_block;
203     uint32_t block_index = fd_table[fd].position / block_size;
204     uint32_t block_offset = fd_table[fd].position % block_size;
205
206     // navigate to the correct block in the chain
207     for (uint32_t i = 0; i < block_index; i++) {
208         if (current_block == 0 || current_block == FAT_EOF) {
209             // unexpected end of chain
210             P_ERRNO = P_EINVAL;
211             return -1;
212         }
213         current_block = fat[current_block];
214     }
215

```

```

216 // now we're at the right block, start reading
217 uint32_t bytes_read = 0;
218
219 while (bytes_read < bytes_to_read) {
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     // seek to the right position in the file
228     if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + block_offset,
229             SEEK_SET) == -1) {
230         P_ERRNO = P_ELSEEK;
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) {
241         P_ERRNO = P_EREAD;
242         // if we already read some data, return that count
243         if (bytes_read > 0) {
244             fd_table[fd].position += bytes_read;
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) {
256         if (current_block == FAT_EOF) {
257             // unexpected end of chain
258             break;
259         }
260         current_block = fat[current_block];
261         block_offset = 0;
262     }
263
264     // if we read less than expected, we might have hit EOF
265     if (read_result < bytes_to_read_now) {
266         break;
267     }
268 }
269
270 // update file position
271 fd_table[fd].position += bytes_read;
272
273 return bytes_read;
274 }

```

4.5.1.6 k_unlink()

```

int k_unlink (
    const char * 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     }
569 }

```

```

570     if (!is_mounted) {
571         P_ERRNO = P_EFS_NOT_MOUNTED;
572         return -1;
573     }
574
575     // check if file is currently open by any process
576     for (int i = 0; i < MAX_FDS; i++) {
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
584     dir_entry_t entry;
585     int file_offset = find_file(fname, &entry);
586     if (file_offset < 0) {
587         P_ERRNO = P_ENOENT;
588         return -1;
589     }
590
591     // 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
604     // free all blocks in the file chain
605     uint16_t current_block = entry.firstBlock;
606     uint16_t next_block;
607
608     while (current_block != FAT_FREE && current_block != FAT_EOF) {
609         next_block = fat[current_block];
610         fat[current_block] = FAT_FREE;
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
281         if (fd == STDOUT_FILENO) {
282             return write(STDOUT_FILENO, str, n);
283         }
284         if (fd == STDERR_FILENO) {
285             return write(STDERR_FILENO, str, n);
286         }
287
288         // validate inputs
289         if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
290             P_ERRNO = P_EBADF;
291             return -1;
292         }
293         if (str == NULL || n < 0) {
294             P_ERRNO = P_EINVAL;

```

```

295     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
311 // create a local buffer for block data
312 char* block_buffer = (char*)malloc(block_size);
313 if (block_buffer == NULL) {
314     P_ERRNO = P_EALLOC;
315     return -1;
316 }
317
318 // calculate initial block position
319 uint32_t block_index = current_position / block_size;
320 uint32_t block_offset = current_position % block_size;
321
322 // if the file doesn't have a first block yet, allocate one
323 if (current_block == 0) {
324     current_block = allocate_block();
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 // navigate to the appropriate block
334 uint16_t prev_block = 0;
335 for (uint32_t i = 0; i < block_index; i++) {
336     if (current_block == 0 || current_block == FAT_EOF ||
337         current_block >= fat_size / 2) {
338         // reached the end of chain prematurely, need to allocate a new block
339         uint16_t new_block = allocate_block();
340         if (new_block == 0) {
341             P_ERRNO = P_EFULL;
342             free(block_buffer);
343             return -1;
344         }
345
346         // update the chain
347         if (prev_block != 0 && prev_block < fat_size / 2) {
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) {
361         P_ERRNO = P_EINVAL;
362         free(block_buffer);
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
370 if (current_block == 0 || current_block == FAT_EOF ||
371     current_block >= fat_size / 2) {
372     if (prev_block != 0 && prev_block < fat_size / 2) {
373         uint16_t new_block = allocate_block();
374         if (new_block == 0) {
375             P_ERRNO = P_EFULL;
376             free(block_buffer);
377             return -1;
378         }
379
380         fat[prev_block] = new_block;
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) {
393     // validate current block
394     if (current_block == 0 || current_block == FAT_EOF ||
395         current_block >= fat_size / 2) {
396         P_ERRNO = P_EINVAL;
397         break;
398     }
399
400     // how much can we write to this block
401     uint32_t space_in_block = block_size - block_offset;
402     uint32_t bytes_to_write = (n - bytes_written) < space_in_block
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
411     if (bytes_to_write < block_size || block_offset > 0) {
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);
420         if (read_result < 0) {
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);
427
428         // seek back to write the modified block
429         if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
430             P_ERRNO = P_ELSEEK;
431             break;
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         }
441     } else {
442         // 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);
449         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
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) {
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();
472         if (new_block == 0) {
473             P_ERRNO = P_EFULL;
474             break;
475         }
476
477         // Update the FAT safely
478         if (current_block < fat_size / 2) {
479             fat[current_block] = new_block;
480         } else {
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 // 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
503 // update the directory entry
504 dir_entry_t entry;
505 int dir_offset = find_file(fd_table[fd].filename, &entry);
506 if (dir_offset >= 0) {
507     entry.size = fd_table[fd].size;
508     entry.mtime = time(NULL);
509
510     if (lseek(fs_fd, dir_offset, SEEK_SET) == -1) {
511         P_ERRNO = P_ELSEEK;
512         return -1;
513     }
514     if (write(fs_fd, &entry, sizeof(entry)) != sizeof(entry)) {
515         P_ERRNO = P_EWRITE;
516         return -1;
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]

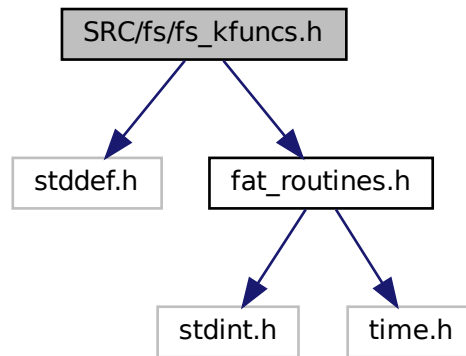
Definition at line 38 of file scheduler.c.

4.6 SRC/fs/fs_kfuncs.h File Reference

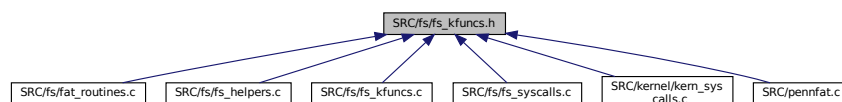
```
#include <stddef.h>
```

```
#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

<i>fd</i>	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.

```

526     {
527     // validate the file descriptor
528     if (fd < 0 || fd >= MAX_FDS) {
529         P_ERRNO = P_EBADF;
530         return -1;
531     }
532
533     // ensure any pending changes are written to disk
534     // update the directory entry with the current file size
535     dir_entry_t entry;
536     int file_offset = find_file(fd_table[fd].filename, &entry);
537
538     if (file_offset >= 0) {
539         // update file size if it changed
540         if (entry.size != fd_table[fd].size) {
541             entry.size = fd_table[fd].size;
542             entry.mtime = time(NULL);
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)) {
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 }
```

4.6.2.2 k_ls()

```

int k_ls (
    const char * 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

<i>filename</i>	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.

```

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;
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
681             if (lseek(fs_fd, fat_size + (current_block - 1) * block_size, SEEK_SET) ==
682                 -1) {
683                 P_ERRNO = P_ELSEEK;
684                 return -1;
685             }
686
687             offset = 0;
688
689             // search current block
690             while (offset < block_size) {
691                 if (read(fs_fd, &dir_entry, sizeof(dir_entry)) != sizeof(dir_entry)) {
692                     P_ERRNO = P_EREAD;
693                     return -1;
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
702                 if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
703                     offset += sizeof(dir_entry);
704                     continue;
705                 }
706
707                 // format permission string
708                 char perm_str[4] = "---";
709                 if (dir_entry.perm & PERM_READ)
710                     perm_str[0] = 'r';
711                 if (dir_entry.perm & PERM_WRITE)
712                     perm_str[1] = 'w';
713                 if (dir_entry.perm & PERM_EXEC)
714                     perm_str[2] = 'x';
715
716                 // format time
717                 struct tm* tm_info = localtime(&dir_entry.mtime);
718                 char time_str[50];
719                 strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);
720
721                 // print entry details
722                 char buffer[128];
723                 int len;
724                 if (dir_entry.firstBlock == 0) {
725                     len = snprintf(buffer, sizeof(buffer), "  -%- %6d %s %s\n",
726                                     perm_str, dir_entry.size, time_str, dir_entry.name);
727                 } else {
728                     len = snprintf(buffer, sizeof(buffer), "%2d -%- %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;
735                     return -1;
736                 }
737
738                 if (k_write(STDOUT_FILENO, buffer, len) != len) {
739                     P_ERRNO = P_EWRITE;
740                     return -1;

```

```

741     }
742
743     offset += sizeof(dir_entry);
744 }
745
746 // 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
752 // no more blocks to search
753 break;
754 }
755 } else {
756     // find and display specific file
757     int file_offset = find_file(filename, &dir_entry);
758     if (file_offset < 0) {
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
769     if (dir_entry.name[0] == 1 || dir_entry.name[0] == 2) {
770         P_ERRNO = P_ENOENT;
771         return -1;
772     }
773
774     // format permission string
775     char perm_str[4] = "---";
776     if (dir_entry.perm & PERM_READ)
777         perm_str[0] = 'r';
778     if (dir_entry.perm & PERM_WRITE)
779         perm_str[1] = 'w';
780     if (dir_entry.perm & PERM_EXEC)
781         perm_str[2] = 'x';
782
783     // format time
784     struct tm* tm_info = localtime(&dir_entry.mtime);
785     char time_str[50];
786     strftime(time_str, sizeof(time_str), "%b %d %H:%M:%S %Y", tm_info);
787
788     // print entry details
789     char buffer[128];
790     int len;
791     if (dir_entry.firstBlock == 0) {
792         len = snprintf(buffer, sizeof(buffer), "  -%- %6d %s %s\n", perm_str,
793             dir_entry.size, time_str, dir_entry.name);
794     } else {
795         len = snprintf(buffer, sizeof(buffer), "%2d -%- %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;
802         return -1;
803     }
804
805     if (k_write(STDOUT_FILENO, buffer, len) != len) {
806         P_ERRNO = P_EWRITE;
807         return -1;
808     }
809 }
810
811 return 0;
812 }

```

4.6.2.3 k_lseek()

```

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

```

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

<i>fd</i>	File descriptor of the open file.
<i>offset</i>	The offset in bytes to set the position to.
<i>whence</i>	How to interpret the offset: <ul style="list-style-type: none"> • SEEK_SET (0): Offset is from the beginning of the file. • SEEK_CUR (1): Offset is from the current position. • SEEK_END (2): Offset is from the end of the file.

Returns

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

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

Repositions the file offset of an open file.

Definition at line 620 of file fs_kfuncs.c.

```

620                                     {
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
628     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:
638             new_position = offset;
639             break;
640         case SEEK_CUR:
641             new_position = fd_table[fd].position + offset;
642             break;
643         case SEEK_END:
644             new_position = fd_table[fd].size + offset;
645             break;
646         default:
647             P_ERRNO = P_EINVAL;
648             return -1;
649     }
650
651     // check if new position is valid
652     if (new_position < 0) {
653         P_ERRNO = P_EINVAL;
654         return -1;
655     }
656
657     // update file position
658     fd_table[fd].position = new_position;
659
660     return new_position;
661 }
```

4.6.2.4 k_open()

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

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

<i>fname</i>	The name of the file to open.
<i>mode</i>	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.

```
31                                     {
32     // validate arguments
33     if (fname == NULL || *fname == '\0') {
34         P_ERRNO = P_EINVAL;
35         return -1;
36     }
37     if ((mode & (F_READ | F_WRITE | F_APPEND)) == 0) {
38         P_ERRNO = P_EINVAL;
39         return -1;
40     }
41
42     // check if the file system is mounted
43     if (!is_mounted) {
44         P_ERRNO = P_EFS_NOT_MOUNTED;
45         return -1;
46     }
47
48     // get a free file descriptor
49     int fd = get_free_fd(fd_table);
50     if (fd < 0) {
51         P_ERRNO = P_EFULL; // no free file descriptors
52         return -1;
53     }
54
55     // check if the file exists
56     dir_entry_t entry;
57     int file_offset = find_file(fname, &entry);
58
59     // file exists
60     if (file_offset >= 0) {
61         // check if the file is already open in write mode by another descriptor
62         if ((mode & (F_WRITE | F_APPEND)) != 0) {
63             for (int i = 0; i < MAX_FDS; i++) {
64                 if (i != fd && fd_table[i].in_use &&
65                     strcmp(fd_table[i].filename, fname) == 0 &&
66                     (fd_table[i].mode & (F_WRITE | F_APPEND)) != 0) {
67                     P_ERRNO = P_EBUSY; // file is already open for writing
68                     return -1;
69                 }
70             }
71         }
72     }
73     // 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] = '\0';
78     fd_table[fd].size = entry.size;
79     fd_table[fd].first_block = entry.firstBlock;
80     fd_table[fd].mode = mode;
81
82     // set the initial position
83     if (mode & F_APPEND) {
84         fd_table[fd].position = entry.size;
85     } else {
86         fd_table[fd].position = 0;
87     }
88
89     // if mode includes F_WRITE and not 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;
93         uint16_t next_block;
94
95         if (block != 0 && block != FAT_EOF) {
96             next_block = fat[block];
97             fat[block] = FAT_EOF; // terminate the chain at the first block
98             block = next_block;
99
100            // free the rest of the chain
101            while (block != 0 && block != FAT_EOF) {
102                next_block = fat[block];
103                fat[block] = FAT_FREE;
104                block = next_block;
105            }
106        }
107
108        // update file size to 0
109        fd_table[fd].size = 0;
110        entry.size = 0;
111        entry.mtime = time(NULL);
112
113        // update the file system with the truncated file
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;
120            return -1;
121        }
122    }
123 } else {
124     // file doesn't exist
125
126     // we can only create it if we are reading the file
127     if (!(mode & F_WRITE)) {
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) {
135         P_ERRNO = P_EFULL;
136         return -1;
137     }
138
139     // create a new file entry
140     if (add_file_entry(fname, 0, first_block, TYPE_REGULAR, PERM_READ_WRITE) ==
141         -1) {
142         // error code already set by add_file_entry
143         fat[first_block] = FAT_FREE;
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] = '\0';
152     fd_table[fd].size = 0;
153     fd_table[fd].first_block = first_block;
154     fd_table[fd].position = 0;
155     fd_table[fd].mode = mode;
156 }
157
158 return fd;
159 }

```

4.6.2.5 k_read()

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

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

<i>fd</i>	File descriptor of the open file.
<i>buf</i>	Buffer to store the read data.
<i>n</i>	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 {
165     // 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) {
168             s_kill(current_running_pcb->pid, P_SIGSTOP);
169         }
170     }
171
172     // handle standard input
173     if (fd == STDIN_FILENO) {
174         return read(STDIN_FILENO, buf, n);
175     }
176
177     // validate inputs
178     if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
179         P_ERRNO = P_EBADF;
180         return -1;
181     }
182     if (buf == NULL || n < 0) {
183         P_ERRNO = P_EINVAL;
184         return -1;
185     }
186     if (n == 0) {
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
195     // determine how many bytes we can actually read
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 // find the block containing the current position
202 uint16_t current_block = fd_table[fd].first_block;
203 uint32_t block_index = fd_table[fd].position / block_size;
204 uint32_t block_offset = fd_table[fd].position % block_size;
205
206 // navigate to the correct block in the chain
207 for (uint32_t i = 0; i < block_index; i++) {
208     if (current_block == 0 || current_block == FAT_EOF) {
209         // unexpected end of chain
210         P_ERRNO = P_EINVAL;
211         return -1;
212     }
213     current_block = fat[current_block];
214 }
215
216 // now we're at the right block, start reading
217 uint32_t bytes_read = 0;
218
219 while (bytes_read < bytes_to_read) {
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     // seek to the right position in the file
228     if (lseek(fs_fd, fat_size + (current_block - 1) * block_size + block_offset,
229             SEEK_SET) == -1) {
230         P_ERRNO = P_ELSEEK;
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) {
241         P_ERRNO = P_EREAD;
242         // if we already read some data, return that count
243         if (bytes_read > 0) {
244             fd_table[fd].position += bytes_read;
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) {
256         if (current_block == FAT_EOF) {
257             // unexpected end of chain
258             break;
259         }
260         current_block = fat[current_block];
261         block_offset = 0;
262     }
263
264     // if we read less than expected, we might have hit EOF
265     if (read_result < bytes_to_read_now) {
266         break;
267     }
268 }
269
270 // update file position
271 fd_table[fd].position += bytes_read;
272
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

<i>fname</i>	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.

```

564
565     if (fname == NULL || *fname == '\\0') {
566         P_ERRNO = P_EINVAL;
567         return -1;
568     }
569
570     if (!is_mounted) {
571         P_ERRNO = P_EFS_NOT_MOUNTED;
572         return -1;
573     }
574
575     // check if file is currently open by any process
576     for (int i = 0; i < MAX_FDS; i++) {
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
584     dir_entry_t entry;
585     int file_offset = find_file(fname, &entry);
586     if (file_offset < 0) {
587         P_ERRNO = P_ENOENT;
588         return -1;
589     }
590
591     // 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
604     // free all blocks in the file chain
605     uint16_t current_block = entry.firstBlock;
606     uint16_t next_block;
607
608     while (current_block != FAT_FREE && current_block != FAT_EOF) {
609         next_block = fat[current_block];
610         fat[current_block] = FAT_FREE;
611         current_block = next_block;
612     }
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

<i>fd</i>	File descriptor of the open file.
<i>str</i>	Buffer containing the data to write.
<i>n</i>	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`.

```
279                                     {
280     // handle standard output and error
281     if (fd == STDOUT_FILENO) {
282         return write(STDOUT_FILENO, str, n);
283     }
284     if (fd == STDERR_FILENO) {
285         return write(STDERR_FILENO, str, n);
286     }
287
288     // validate inputs
289     if (fd < 0 || fd >= MAX_FDS || !fd_table[fd].in_use) {
290         P_ERRNO = P_EBADF;
291         return -1;
292     }
293     if (str == NULL || n < 0) {
294         P_ERRNO = P_EINVAL;
295         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
311     // create a local buffer for block data
312     char* block_buffer = (char*)malloc(block_size);
313     if (block_buffer == NULL) {
314         P_ERRNO = P_EMALLOC;
315         return -1;
316     }
317 }
```

```

318 // calculate initial block position
319 uint32_t block_index = current_position / block_size;
320 uint32_t block_offset = current_position % block_size;
321
322 // if the file doesn't have a first block yet, allocate one
323 if (current_block == 0) {
324     current_block = allocate_block();
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 // navigate to the appropriate block
334 uint16_t prev_block = 0;
335 for (uint32_t i = 0; i < block_index; i++) {
336     if (current_block == 0 || current_block == FAT_EOF ||
337         current_block >= fat_size / 2) {
338         // reached the end of chain prematurely, need to allocate a new block
339         uint16_t new_block = allocate_block();
340         if (new_block == 0) {
341             P_ERRNO = P_EFULL;
342             free(block_buffer);
343             return -1;
344         }
345
346         // update the chain
347         if (prev_block != 0 && prev_block < fat_size / 2) {
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) {
361         P_ERRNO = P_EINVAL;
362         free(block_buffer);
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
370 if (current_block == 0 || current_block == FAT_EOF ||
371     current_block >= fat_size / 2) {
372     if (prev_block != 0 && prev_block < fat_size / 2) {
373         uint16_t new_block = allocate_block();
374         if (new_block == 0) {
375             P_ERRNO = P_EFULL;
376             free(block_buffer);
377             return -1;
378         }
379
380         fat[prev_block] = new_block;
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) {
393     // validate current block
394     if (current_block == 0 || current_block == FAT_EOF ||
395         current_block >= fat_size / 2) {
396         P_ERRNO = P_EINVAL;
397         break;
398     }
399
400     // how much can we write to this block
401     uint32_t space_in_block = block_size - block_offset;
402     uint32_t bytes_to_write = (n - bytes_written) < space_in_block
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
411 if (bytes_to_write < block_size || block_offset > 0) {
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);
420     if (read_result < 0) {
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);
427
428     // seek back to write the modified block
429     if (lseek(fs_fd, block_position, SEEK_SET) == -1) {
430         P_ERRNO = P_ELSEEK;
431         break;
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     }
441 } else {
442     // 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);
449     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
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) {
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();
472         if (new_block == 0) {
473             P_ERRNO = P_EFULL;
474             break;
475         }
476
477         // Update the FAT safely
478         if (current_block < fat_size / 2) {
479             fat[current_block] = new_block;
480         } else {
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 // 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
503     dir_entry_t entry;
504     int dir_offset = find_file(fd_table[fd].filename, &entry);
505     if (dir_offset >= 0) {
506         entry.size = fd_table[fd].size;
507         entry.mtime = time(NULL);
508
509         if (lseek(fs_fd, dir_offset, SEEK_SET) == -1) {
510             P_ERRNO = P_ELSEEK;
511             return -1;
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 }

```

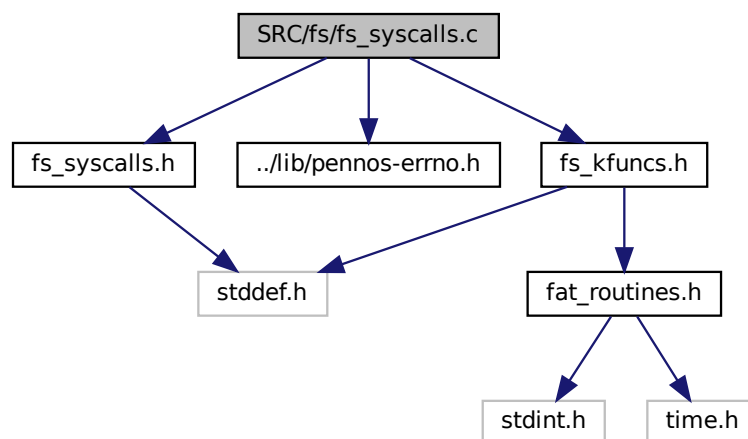
4.7 SRC/fs/fs_syscalls.c File Reference

```

#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`.

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

4.7.1.2 `s_ls()`

```
int s_ls (  
    const char * filename )
```

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()

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

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()

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

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()

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

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.

```
24 {
25     return k_read(fd, buf, n);
26 }
```

4.7.1.6 s_unlink()

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

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.

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

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.

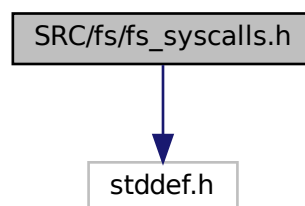
Definition at line 33 of file fs_syscalls.c.

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

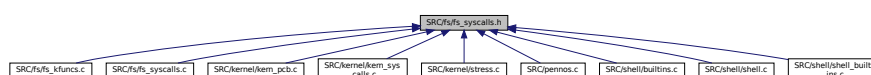
4.8 SRC/fs/fs_syscalls.h File Reference

```
#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 (
    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

<i>fd</i>	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 (
    const char * 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

<i>filename</i>	The name of the file to get information about, or <code>NULL</code> to list all files.
-----------------	--

Returns

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

- P_ENOENT: The specified file does not exist.

Lists files in the current directory or displays file information.

This is a wrapper around the kernel function k_ls.

Definition at line 69 of file fs_syscalls.c.

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

4.8.2.3 s_lseek()

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

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

<i>fd</i>	The file descriptor of an open file.
<i>offset</i>	The offset in bytes.
<i>whence</i>	Specifies the reference position: <ul style="list-style-type: none">• 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()

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

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

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

Returns

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

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

Opens a file with the specified access mode.

This is a wrapper around the kernel function k_open.

Definition at line 15 of file fs_syscalls.c.

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

4.8.2.5 s_read()

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

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

<i>fd</i>	The file descriptor of an open file.
<i>n</i>	The maximum number of bytes to read.
<i>buf</i>	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.

```
24 {  
25     return k_read(fd, buf, n);  
26 }
```

4.8.2.6 s_unlink()

```
int s_unlink (  
    const char * 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

<i>fname</i>	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

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

Returns

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

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

Writes data to an open file.

This is a wrapper around the kernel function *k_write*.

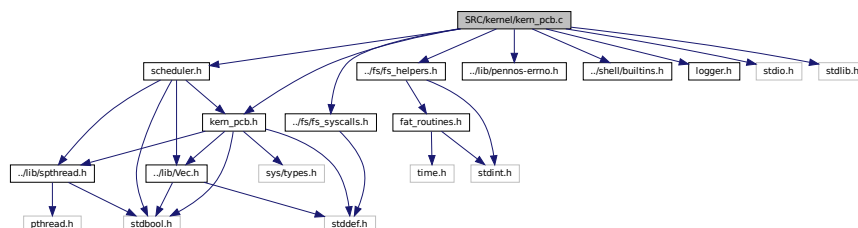
Definition at line 33 of file *fs_syscalls.c*.

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

4.9 SRC/kernel/kern_pcb.c File Reference

```
#include "kern_pcb.h"
#include "../fs/fs_helpers.h"
#include "../fs/fs_syscalls.h"
#include "../lib/pennos-errno.h"
#include "../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_pcb`s
- `pcb_t *` `current_running_pcb`

4.9.1 Function Documentation

4.9.1.1 `create_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.

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.

```
46     {
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     }
52
53     ret_pcb->pid = pid;
54     ret_pcb->par_pid = par_pid;
55     ret_pcb->priority = priority;
56     ret_pcb->process_state = 'R'; // running by default
57     ret_pcb->input_fd = input_fd;
58     ret_pcb->output_fd = output_fd;
59     ret_pcb->process_status = 0; // default status
60
61     ret_pcb->child_pcb = vec_new(0, NULL); // NULL deconstructor prevents
62                                         // double free
63
64     for (int i = 0; i < 3; i++) {
65         ret_pcb->signals[i] = false;
66     }
67
68     ret_pcb->is_sleeping = false;
69     ret_pcb->time_to_wake = -1; // default to not sleeping
70
71     return ret_pcb;
72 }
```

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.

```
30 {
31     pcb_t* casted_pcb = (pcb_t*)pcb;
32
33     free(casted_pcb->cmd_str);
34     vec_destroy(&casted_pcb->child_pcbs); // will free any remaining
35                                           // children too!
36     free(casted_pcb);
37 }
```

4.9.1.3 k_proc_cleanup()

```
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.

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 {
153     // if proc has parent (i.e. isn't init) then remove it from parent's child
154     // list
155     pcb_t* par_pcb = get_pcb_in_queue(&current_pcbs, proc->par_pid);
156     if (par_pcb != NULL) {
157         remove_child_in_parent(par_pcb, proc);
158     } else {
159         P_ERRNO = P_ENULL;
160         return;
161     }
162
163     // if proc has children, remove them and assign them to init parent
164     if (vec_len(&proc->child_pcbs) > 0) {
165         // retrieve the init process
166         pcb_t* init_pcb =
167             get_pcb_in_queue(&current_pcbs, 1); // init process has pid 1
168
169         while (vec_len(&proc->child_pcbs) > 0) {
170             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             curr_child->par_pid = 1; // update parent to init (pid 1)
174             log_generic_event('O', curr_child->pid, curr_child->priority,
175                             curr_child->cmd_str);
176         }
177     }
178
179     // decr reference counts + close files if necessary
180     for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
181         if (proc->fd_table[i] != -1) {
182             if (decrement_fd_ref_count(proc->fd_table[i]) == 0) {
183                 if (s_close(proc->fd_table[i]) == -1) {
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 }

```

4.9.1.4 k_proc_create()

```

pcb_t* k_proc_create (
    pcb_t * parent,
    int priority )

```

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.

```

95 {
96     if (parent == NULL) { // init creation case
97         pcb_t* init = create_pcb(1, 0, 0, 0, 1);
98         if (init == NULL) {
99             P_ERRNO = P_ENULL;
100             return NULL;
101         }
102         init->fd_table[0] = STDIN_FILENO;
103         init->fd_table[1] = STDOUT_FILENO;
104         init->fd_table[2] = STDERR_FILENO;
105         for (int i = 3; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
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_pcb, init);
116         return init;
117     }
118
119     pcb_t* child = create_pcb(next_pid++, parent->pid, priority, parent->input_fd,
120                             parent->output_fd);
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++) {
128         child->fd_table[i] = parent->fd_table[i];
129     }
130
131     for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
132         if (child->fd_table[i] != -1) {
133             increment_fd_ref_count(child->fd_table[i]);
134         }
135     }
136
137     // update parent as needed
138     vec_push_back(&parent->child_pcb, child);
139
140     // add to appropriate queue
141     put_pcb_into_correct_queue(child);
142     vec_push_back(&current_pcb, child);
143
144     return child;
145 }

```

4.9.1.5 remove_child_in_parent()

```
void remove_child_in_parent (
    pcb_t * parent,
    pcb_t * child )
```

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`.

```
77
78     for (int i = 0; i < vec_len(&parent->child_pcb); i++) {
79         pcb_t* curr_child = (pcb_t*)vec_get(&parent->child_pcb, i);
80         if (curr_child->pid == child->pid) {
81             vec_erase_no_deletor(&parent->child_pcb, i);
82             return;
83         }
84     }
85 }
```

4.9.2 Variable Documentation

4.9.2.1 current_pcb

```
Vec current_pcb [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`.

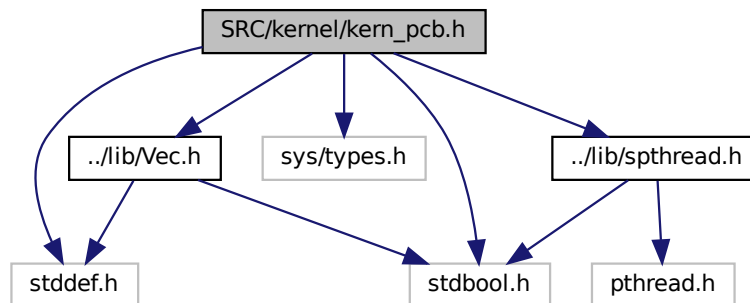
4.9.2.3 next_pid

```
int next_pid = 2
```

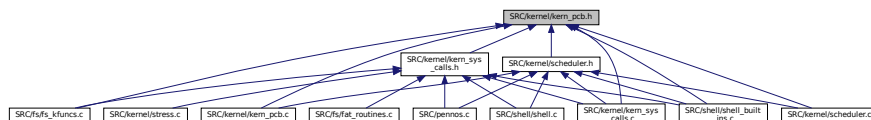
Definition at line 17 of file `kern_pcb.c`.

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()

```
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.

Parameters

<i>pid</i>	the new process id
<i>par_pid</i>	the parent process id
<i>priority</i>	the priority level (0,1,2)
<i>input_fd</i>	input fd
<i>output_fd</i>	output 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.

```

46  {
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  }
52
53  ret_pcb->pid = pid;
54  ret_pcb->par_pid = par_pid;
55  ret_pcb->priority = priority;
56  ret_pcb->process_state = 'R'; // running by default
57  ret_pcb->input_fd = input_fd;
58  ret_pcb->output_fd = output_fd;
59  ret_pcb->process_status = 0; // default status
60
61  ret_pcb->child_pcb = vec_new(0, NULL); // NULL deconstructor prevents
62                                         // double free
63
64  for (int i = 0; i < 3; i++) {
65      ret_pcb->signals[i] = false;
66  }
67
68  ret_pcb->is_sleeping = false;
69  ret_pcb->time_to_wake = -1; // default to not sleeping
70
71  return ret_pcb;
72 }
```

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

<i>pcb</i>	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.

```

30      {
31  pcb_t* casted_pcb = (pcb_t*)pcb;
32
33  free(casted_pcb->cmd_str);
34  vec_destroy(&casted_pcb->child_pcbs); // will free any remaining
35                                     // children too!
36  free(casted_pcb);
37 }

```

4.10.3.3 k_proc_cleanup()

```

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.

Parameters

<i>proc</i>	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      {
153  // if proc has parent (i.e. isn't init) then remove it from parent's child
154  // list
155  pcb_t* par_pcb = get_pcb_in_queue(&current_pcbs, proc->par_pid);
156  if (par_pcb != NULL) {
157      remove_child_in_parent(par_pcb, proc);
158  } else {
159      P_ERRNO = P_ENULL;
160      return;
161  }
162
163  // if proc has children, remove them and assign them to init parent
164  if (vec_len(&proc->child_pcbs) > 0) {
165      // retrieve the init process
166      pcb_t* init_pcb =
167          get_pcb_in_queue(&current_pcbs, 1); // init process has pid 1
168
169      while (vec_len(&proc->child_pcbs) > 0) {
170          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          curr_child->par_pid = 1; // update parent to init (pid 1)
174          log_generic_event('O', curr_child->pid, curr_child->priority,
175                          curr_child->cmd_str);
176      }
177  }
178
179  // decr reference counts + close files if necessary
180  for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
181      if (proc->fd_table[i] != -1) {
182          if (decrement_fd_ref_count(proc->fd_table[i]) == 0) {
183              if (s_close(proc->fd_table[i]) == -1) {
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()

```

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.

Parameters

<i>parent</i>	a pointer to the parent pcb
<i>priority</i>	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.

```

95  {
96  if (parent == NULL) { // init creation case
97  pcb_t* init = create_pcb(1, 0, 0, 0, 1);
98  if (init == NULL) {
99  P_ERRNO = P_ENULL;
100  return NULL;
101  }
102  init->fd_table[0] = STDIN_FILENO;
103  init->fd_table[1] = STDOUT_FILENO;
104  init->fd_table[2] = STDERR_FILENO;
105  for (int i = 3; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
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
119  pcb_t* child = create_pcb(next_pid++, parent->pid, priority, parent->input_fd,
120  parent->output_fd);
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++) {
128  child->fd_table[i] = parent->fd_table[i];
129  }
130
131  for (int i = 0; i < FILE_DESCRIPTOR_TABLE_SIZE; i++) {
132  if (child->fd_table[i] != -1) {
133  increment_fd_ref_count(child->fd_table[i]);
134  }

```

```

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

```

4.10.3.5 remove_child_in_parent()

```

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.

Parameters

<i>parent</i>	a ptr to the parent pcb with the child list
<i>child</i>	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`.

```

77
78 for (int i = 0; i < vec_len(&parent->child_pcb); i++) {
79     pcb_t* curr_child = (pcb_t*)vec_get(&parent->child_pcb, i);
80     if (curr_child->pid == child->pid) {
81         vec_erase_no_deletor(&parent->child_pcb, i);
82         return;
83     }
84 }
85 }

```

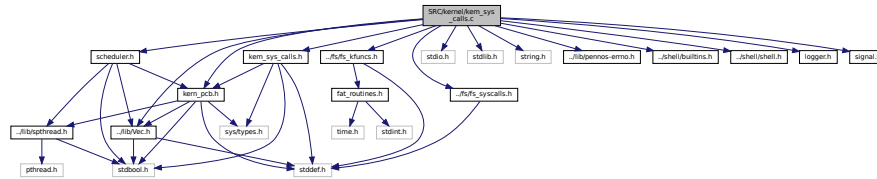
4.11 SRC/kernel/kern_sys_calls.c File Reference

```

#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 dependency graph for kern_sys_calls.c:



- 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".

- `Vec zero_priority_queue`
- `Vec one_priority_queue`
- `Vec two_priority_queue`
- `Vec zombie_queue`
- `Vec sleep_blocked_queue`
- `Vec current_pcb`
- `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()

```
void delete_from_explicit_queue (
    Vec * queue_to_delete_from,
    int pid )
```

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()

```
void delete_from_queue (
    int queue_id,
    int pid )
```

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()

```
int determine_index_in_queue (
    Vec * queue,
    int pid )
```

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.

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

4.11.1.4 init_func()

```
void* init_func (
    void * input )
```

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.

```
118 {
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) {
124         int status;
125         s_waitpid(-1, &status, false);
126     }
127
128     return NULL; // should never reach
129 }
```

4.11.1.5 move_pcb_correct_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.

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.

```

57                                     {
58     Vec* prev_queue;
59     Vec* new_queue;
60
61     if (prev_priority == 0) {
62         prev_queue = &zero_priority_queue;
63     } else if (prev_priority == 1) {
64         prev_queue = &one_priority_queue;
65     } else {
66         prev_queue = &two_priority_queue;
67     }
68
69     if (new_priority == 0) {
70         new_queue = &zero_priority_queue;
71     } else if (new_priority == 1) {
72         new_queue = &one_priority_queue;
73     } else {
74         new_queue = &two_priority_queue;
75     }
76
77     // delete from prev_queue, if it's present at all
78     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);
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.

```

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

4.11.1.7 s_echo()

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

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

Parameters

<i>arg</i>	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     int i = 1; // words after "echo"
369     while (argv[i] != NULL) { // while the arg isn't NULL
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
380     if (s_write(current_running_pcb->output_fd, "\n", 1) == -1) {
381         u_perror("s_write error");
382     }
383     return NULL;
384 }

```

4.11.1.8 s_exit()

```

void s_exit (
    void )

```

Exits the current process and cleans up its resources.

Unconditionally exit the calling process.

Definition at line 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     log_generic_event('E', current_running_pcb->pid,
303                     current_running_pcb->priority,
304                     current_running_pcb->cmd_str);
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,
310                     current_running_pcb->cmd_str);
311 }

```

4.11.1.9 s_kill()

```

int s_kill (
    pid_t pid,
    int signal )

```

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.

```

282     {
283     pcb_t* pcb_with_pid = get_pcb_in_queue(&current_pcb, 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 }

```

4.11.1.10 s_nice()

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

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
321     pcb_t* curr_pcb = get_pcb_in_queue(&current_pcbs, pid);
322     if (curr_pcb != NULL) { // found + exists
323     move_pcb_correct_queue(curr_pcb->priority, priority, curr_pcb);
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
330 }
```

4.11.1.11 s_ps()

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

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

Parameters

<i>arg</i>	the pass along arguments to the u_ps function
------------	---

Returns

NULL, dummy return value

Definition at line 389 of file kern_sys_calls.c.

```
389     {
390     char pid_top[] = "PID\tPPID\tPRI\tSTAT\tCMD\n";
391     if (s_write(current_running_pcb->output_fd, pid_top, strlen(pid_top)) == -1) {
392     u_perror("s_write error");
393     }
394     for (int i = 0; i < vec_len(&current_pcbs); i++) {
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,
399     curr_pcb->cmd_str);
400     if (s_write(current_running_pcb->output_fd, buffer, strlen(buffer)) == -1) {
401     u_perror("s_write error");
402     }
403     }
404     return NULL;
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.

```
335                                     {
336     if (ticks <= 0) {
337         P_ERRNO = P_EINVAL;
338         return;
339     }
340
341     // block current process, set state to sleep
342     current_running_pcb->process_state = 'B';
343     current_running_pcb->is_sleeping = true;
344     current_running_pcb->time_to_wake = tick_counter + ticks;
345     log_generic_event('B', current_running_pcb->pid,
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
350         perror("Error in pthread_suspend in s_sleep call");
351     }
352 }
```

4.11.1.13 s_spawn()

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

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                                     {
166     pcb_t* child;
167     if (strcmp(argv[0], "shell") == 0) {
168         child = k_proc_create(current_running_pcb, 0);
169     } else {
170         child = k_proc_create(current_running_pcb, 1);
171     }
172
173     if (child == NULL) {
174         P_ERRNO = P_ENULL;
175         return -1;
176     }
177
178     pthread_t thread_handle;
179
180     if (pthread_create(&thread_handle, NULL, func, argv) != 0) {
181         perror("Error in pthread_create in s_spawn call");
182     }
183
184     child->cmd_str = strdup(argv[0]);
185     child->thread_handle = thread_handle;
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 }
```



```

229     return child->pid;
230 }
231 }
232
233 // If nohang is true, return immediately if no child has exited
234 if (nohang) {
235     return 0;
236 }
237
238 // Block the parent until a child exits
239 delete_from_queue(parent->priority, parent->pid);
240 parent->process_state = 'B';
241 log_generic_event('B', parent->pid, parent->priority, parent->cmd_str);
242
243 while (true) {
244     // Scan the zombie queue first for terminated children.
245     for (int i = 0; i < vec_len(&zombie_queue); i++) {
246         pcb_t* child = vec_get(&zombie_queue, i);
247         if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
248             if (wstatus != NULL) {
249                 *wstatus = child->process_status;
250             }
251             log_generic_event('W', child->pid, child->priority, child->cmd_str);
252             vec_erase_no_deletor(&zombie_queue, i);
253             delete_from_explicit_queue(&parent->child_pcb, child->pid);
254             k_proc_cleanup(child);
255             return child->pid;
256         }
257     }
258
259     // scan children of current running process for non-terminated state changes
260     for (int i = 0; i < vec_len(&parent->child_pcb); i++) {
261         pcb_t* child = vec_get(&parent->child_pcb, i);
262         if ((pid == -1 || child->pid == pid) &&
263             (child->process_status == 21 ||
264              child->process_status == 23)) { // signaled
265             if (wstatus != NULL) {
266                 *wstatus = child->process_status;
267             }
268             log_generic_event('W', child->pid, child->priority, child->cmd_str);
269             child->process_status = 0; // reset status
270             return child->pid;
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_pcb

Vec current_pcb [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.

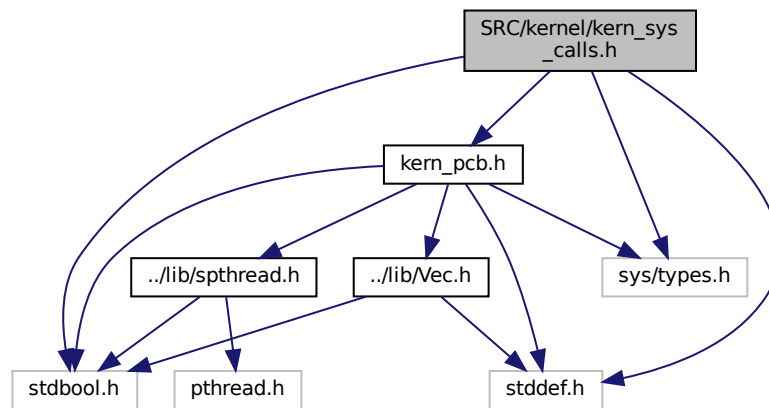
4.11.2.9 zombie_queue

`Vec zombie_queue [extern]`

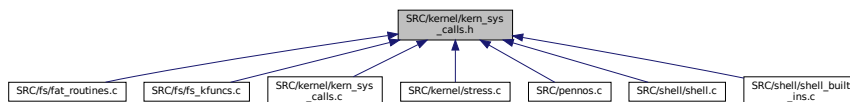
Definition at line 27 of file scheduler.c.

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()

```
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.

Parameters

<code>queue_to_delete_from</code>	ptr to Vec* queue to delete from
<code>pid</code>	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) {
111         vec_erase_no_deletor(queue_to_delete_from, index);
112     }
113 }
```

4.12.1.2 delete_from_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).

Parameters

<i>queue_id</i>	An integer representing the queue: 0 for zero_priority_queue, 1 for one_priority_queue, or 2 for two_priority_queue.
<i>pid</i>	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.

```

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 }
```

4.12.1.3 determine_index_in_queue()

```

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.

Parameters

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

Returns

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

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

Definition at line 40 of file kern_sys_calls.c.

```

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

4.12.1.4 init_func()

```

void* init_func (
    void * input )
```

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

Parameters

<i>input</i>	unused but needed for typing reasons
--------------	--------------------------------------

Returns

irrelevant 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.

```

118      {
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) {
124      int status;
125      s_waitpid(-1, &status, false);
126  }
127
128  return NULL; // should never reach
129  }
```

4.12.1.5 move_pcb_correct_queue()

```

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.

Parameters

<i>prev_priority</i>	thread's previous priority
<i>new_priority</i>	thread's new priority
<i>curr_pcb</i>	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*.

```

57                                     {
58     Vec* prev_queue;
59     Vec* new_queue;
60
61     if (prev_priority == 0) {
62         prev_queue = &zero_priority_queue;
63     } else if (prev_priority == 1) {
64         prev_queue = &one_priority_queue;
65     } else {
66         prev_queue = &two_priority_queue;
67     }
68
69     if (new_priority == 0) {
70         new_queue = &zero_priority_queue;
71     } else if (new_priority == 1) {
72         new_queue = &one_priority_queue;
73     } else {
74         new_queue = &two_priority_queue;
75     }
76
77     // delete from prev_queue, if it's present at all
78     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);
84 }
```

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_pcb, 1));
160 }
```

4.12.1.7 s_echo()

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

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

Parameters

<i>arg</i>	the pass along arguments to the <code>u_echo</code> 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     int i = 1; // words after "echo"
369     while (argv[i] != NULL) { // while the arg isn't NULL
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
380     if (s_write(current_running_pcb->output_fd, "\n", 1) == -1) {
381         u_perror("s_write error");
382     }
383     return NULL;
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     log_generic_event('E', current_running_pcb->pid,
303                     current_running_pcb->priority,
304                     current_running_pcb->cmd_str);
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,
310                     current_running_pcb->cmd_str);
311 }
```

4.12.1.9 `s_kill()`

```

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

Send a signal to a particular process.

Parameters

<i>pid</i>	Process ID of the target proces.
<i>signal</i>	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_pcb, 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

<i>pid</i>	Process ID of the target thread.
<i>priority</i>	The new priority 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     {
317     if (priority < 0 || priority > 2) { // error check
318         return -1;
319     }
320
321     pcb_t* curr_pcb = get_pcb_in_queue(&current_pcb, pid);
322     if (curr_pcb != NULL) { // found + exists
323         move_pcb_correct_queue(curr_pcb->priority, priority, curr_pcb);
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
330 }
```

4.12.1.11 s_ps()

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

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

Parameters

<i>arg</i>	the pass along arguments to the u_ps function
------------	---

Returns

NULL, dummy return value

Definition at line 389 of file kern_sys_calls.c.

```
389     {
390     char pid_top[] = "PID\|tPPID\|tPRI\|tSTAT\|tCMD\|n";
391     if (s_write(current_running_pcb->output_fd, pid_top, strlen(pid_top)) == -1) {
392         u_perror("s_write error");
393     }
394     for (int i = 0; i < vec_len(&current_pcb); i++) {
395         pcb_t* curr_pcb = (pcb_t*)vec_get(&current_pcb, 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,
399             curr_pcb->cmd_str);
400         if (s_write(current_running_pcb->output_fd, buffer, strlen(buffer)) == -1) {
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

<i>ticks</i>	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.

```
335     {
336     if (ticks <= 0) {
337         P_ERRNO = P_EINVAL;
338         return;
339     }
```

```

340
341 // block current process, set state to sleep
342 current_running_pcb->process_state = 'B';
343 current_running_pcb->is_sleeping = true;
344 current_running_pcb->time_to_wake = tick_counter + ticks;
345 log_generic_event('B', current_running_pcb->pid,
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
350     perror("Error in pthread_suspend in s_sleep call");
351 }
352 }

```

4.12.1.13 s_spawn()

```

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

```

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

Parameters

<i>func</i>	Function to be executed by the child process.
<i>argv</i>	Null-terminated array of args, including the command name as <code>argv[0]</code> .
<i>fd0</i>	Input file descriptor.
<i>fd1</i>	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 } else {
170     child = k_proc_create(current_running_pcb, 1);
171 }
172
173 if (child == NULL) {
174     P_ERRNO = P_ENULL;
175     return -1;
176 }
177
178 pthread_t thread_handle;
179
180 if (pthread_create(&thread_handle, NULL, func, argv) != 0) {
181     perror("Error in pthread_create in s_spawn call");
182 }
183
184 child->cmd_str = strdup(argv[0]);
185 child->thread_handle = thread_handle;
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 }

```

4.12.1.14 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`.

```

138     {
139     pcb_t* init = k_proc_create(NULL, 0);
140     if (init == NULL) {
141         P_ERRNO = P_ENULL;
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");
151     init->thread_handle = thread_handle;
152     return init->pid;
153 }

```

4.12.1.15 s_waitpid()

```

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

```

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

Parameters

<i>pid</i>	Process ID of the child to wait for.
<i>wstatus</i>	Pointer to an integer variable where the status will be stored.
<i>nohang</i>	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`.

```

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;
207         for (int i = 0; i < vec_len(&current_pcb); i++) {
208             pcb_t* child = vec_get(&current_pcb, i);
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         // Scan the zombie queue first for terminated children.
219         for (int i = 0; i < vec_len(&zombie_queue); i++) {
220             pcb_t* child = vec_get(&zombie_queue, i);
221             if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
222                 if (wstatus != NULL) {
223                     *wstatus = child->process_status;
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_pcb, child->pid);
228                 k_proc_cleanup(child);
229                 return child->pid;
230             }
231         }
232
233         // If nohang is true, return immediately if no child has exited
234         if (nohang) {
235             return 0;
236         }
237
238         // Block the parent until a child exits
239         delete_from_queue(parent->priority, parent->pid);
240         parent->process_state = 'B';
241         log_generic_event('B', parent->pid, parent->priority, parent->cmd_str);
242
243         while (true) {
244             // Scan the zombie queue first for terminated children.
245             for (int i = 0; i < vec_len(&zombie_queue); i++) {
246                 pcb_t* child = vec_get(&zombie_queue, i);
247                 if ((pid == -1 || child->pid == pid) && child->par_pid == parent->pid) {
248                     if (wstatus != NULL) {
249                         *wstatus = child->process_status;
250                     }
251                     log_generic_event('W', child->pid, child->priority, child->cmd_str);
252                     vec_erase_no_deletor(&zombie_queue, i);
253                     delete_from_explicit_queue(&parent->child_pcb, child->pid);
254                     k_proc_cleanup(child);
255                     return child->pid;
256                 }
257             }
258
259             // scan children of current running process for non-terminated state changes
260             for (int i = 0; i < vec_len(&parent->child_pcb); i++) {
261                 pcb_t* child = vec_get(&parent->child_pcb, i);
262                 if ((pid == -1 || child->pid == pid) &&
263                     (child->process_status == 21 ||
264                      child->process_status == 23)) { // signaled
265                     if (wstatus != NULL) {
266                         *wstatus = child->process_status;
267                     }
268                     log_generic_event('W', child->pid, child->priority, child->cmd_str);
269                     child->process_status = 0; // reset status
270                     return child->pid;
271                 }
272             }
273         }
274     }

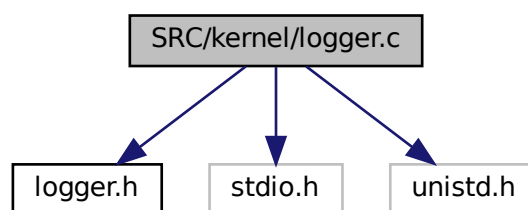
```

```
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()

```
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.

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.

```

23                                     {
24     char* operation;
25
26     switch (event_type) {
27     case 'C':
28         operation = "CREATE";
29         break;
30     case 'S':
31         operation = "SIGNALLED";
32         break;
33     case 'E':
34         operation = "EXITED";
35         break;
36     case 'Z':
37         operation = "ZOMBIE";
38         break;
39     case 'O':
40         operation = "ORPHAN";
41         break;
42     case 'W':
43         operation = "WAITED";
44         break;
45     case 'B':
46         operation = "BLOCKED";
47         break;
48     case 'U':
49         operation = "UNBLOCKED";
50         break;
51     case 's':
52         operation = "STOPPED";
53         break;
54     default:
55         operation = "CONTINUED";
56         break;
57     }
58
59     char buffer[200];
60     int str_len =
61         snprintf(buffer, sizeof(buffer), "[%d]\t%s\t%d\t%d\t%s\n", tick_counter,
62                 operation, pid, nice_value, process_name);
63     if (write(log_fd, buffer, str_len) == -1) {
64         perror("error in writing to the log file for generic event");
65     }
66 }

```

4.13.1.2 log_nice_event()

```

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

```

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.

```

74                                     {
75     char buffer[200];
76     int str_len =
77         snprintf(buffer, sizeof(buffer), "[%d]\tNICE\t%d\t%d\t%d\t%s\n",
78                 tick_counter, pid, old_nice_value, new_nice_value, process_name);
79     if (write(log_fd, buffer, str_len) == -1) {
80         perror("error in writing to the log file for nice event");
81     }
82 }

```

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.

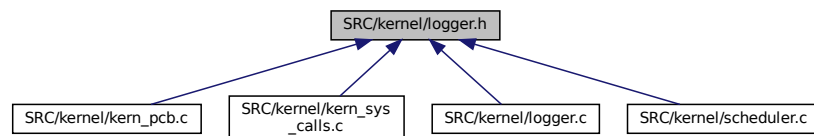
Logs a scheduling event i.e. the scheduling of a process for this clock tick.

Definition at line 8 of file logger.c.

```
8                                     {
9   char buffer[200];
10  int str_len = snprintf(buffer, sizeof(buffer), "[%d]\tSCHEDULE\t%d\t%d\t%s\n",
11                        tick_counter, pid, queue_num, process_name);
12  if (write(log_fd, buffer, str_len) == -1) {
13      perror("error in writing to the log file for scheduling event");
14  }
15 }
```

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()

```
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)

Parameters

<i>event_type</i>	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)
<i>pid</i>	process pid
<i>nice_value</i>	process nice value
<i>process_name</i>	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.

```
23
24     char* operation;
25
26     switch (event_type) {
27         case 'C':
28             operation = "CREATE";
29             break;
30         case 'S':
31             operation = "SIGNALED";
32             break;
33         case 'E':
34             operation = "EXITED";
35             break;
36         case 'Z':
37             operation = "ZOMBIE";
38             break;
39         case 'O':
40             operation = "ORPHAN";
41             break;
42         case 'W':
43             operation = "WAITED";
44             break;
45         case 'B':
46             operation = "BLOCKED";
47             break;
48         case 'U':
49             operation = "UNBLOCKED";
50             break;
51         case 's':
52             operation = "STOPPED";
53             break;
54         default:
55             operation = "CONTINUED";
56             break;
```

```

57  }
58
59  char buffer[200];
60  int str_len =
61      snprintf(buffer, sizeof(buffer), "[%d]\t%s\t%d\t%d\t%s\n", tick_counter,
62                  operation, pid, nice_value, process_name);
63  if (write(log_fd, buffer, str_len) == -1) {
64      perror("error in writing to the log file for generic event");
65  }
66 }

```

4.14.1.2 log_nice_event()

```

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.

Parameters

<i>pid</i>	process pid
<i>old_nice_value</i>	old nice value
<i>new_nice_value</i>	new nice value
<i>process_name</i>	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.

```

74                                     {
75  char buffer[200];
76  int str_len =
77      snprintf(buffer, sizeof(buffer), "[%d]\tNICE\t%d\t%d\t%d\t%s\n",
78                  tick_counter, pid, old_nice_value, new_nice_value, process_name);
79  if (write(log_fd, buffer, str_len) == -1) {
80      perror("error in writing to the log file for nice event");
81  }
82 }

```

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

<i>pid</i>	pid of the process being scheduled
<i>queue_num</i>	the priority queue num of the process
<i>process_name</i>	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.

```

8                                     {
9   char buffer[200];
10  int str_len = snprintf(buffer, sizeof(buffer), "[%d]\tSCHEDULE\t%d\t%d\t%s\n",
11                        tick_counter, pid, queue_num, process_name);
12  if (write(log_fd, buffer, str_len) == -1) {
13      perror("error in writing to the log file for scheduling event");
14  }
15 }
```

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

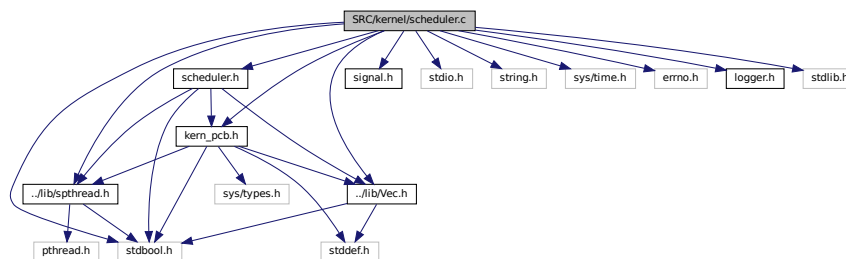
```
int tick_counter [extern]
```

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 "kern_pcb.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_pcb`
- 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()

```
void alarm_handler (  
    int signum )
```

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()

```
bool child_in_zombie_queue (
    pcb_t * parent )
```

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.

```
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 }
```

4.15.1.3 child_with_changed_process_status()

```
bool child_with_changed_process_status (
    pcb_t * parent )
```

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.

```
215                                     {
216     for (int i = 0; i < vec_len(&current_pcb); i++) {
217         pcb_t* child = vec_get(&current_pcb, i);
218         if (child->par_pid == parent->pid && child->process_status != 0) {
219             return true;
220         }
221     }
222     return false;
223 }
```

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_pcb);
182 }
```

4.15.1.5 delete_process_from_all_queues_except_current()

```
void delete_process_from_all_queues_except_current (
    pcb_t * pcb )
```

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 {
169     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()

```
void delete_process_from_particular_queue (
    pcb_t * pcb,
    Vec * 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 implementation 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++) {
157         pcb_t* curr_pcb = vec_get(queue, i);
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.

```
66 {
67     vec_destroy(&zero_priority_queue);
68     vec_destroy(&one_priority_queue);
69     vec_destroy(&two_priority_queue);
70     vec_destroy(&zombie_queue);
71     vec_destroy(&sleep_blocked_queue);
72     vec_destroy(&current_pcb);
73 }
```

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 probabilities. 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.

```
83     {
84     // 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     }
89
90     int priorities_attempted = 0;
91     while (priorities_attempted < 19) {
92         int curr_pri = det_priorities_arr[curr_priority_arr_index];
93         curr_priority_arr_index = (curr_priority_arr_index + 1) % 19;
94         if (curr_pri == 0 && !vec_is_empty(&zero_priority_queue)) {
95             priorities_attempted++;
96             return 0;
97         } else if (curr_pri == 1 && !vec_is_empty(&one_priority_queue)) {
98             priorities_attempted++;
99             return 1;
100        } else if (curr_pri == 2 && !vec_is_empty(&two_priority_queue)) {
101            priorities_attempted++;
102            return 2;
103        }
104    }
105
106    return -1; // should never reach
107 }
```

4.15.1.9 get_next_pcb()

```
pcb_t* get_next_pcb (
    int priority )
```

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


4.15.1.10 get_pcb_in_queue()

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

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.

```
187
188     for (int i = 0; i < vec_len(queue); i++) {
189         pcb_t* curr_pcb = vec_get(queue, i);
190         if (curr_pcb->pid == pid) {
191             return curr_pcb;
192         }
193     }
194
195     return NULL;
196 }
```

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) {
237         case 0: // P_SIGSTOP
238             if (pcb->process_state == 'R' || pcb->process_state == 'B') {
239                 pcb->process_state = 'S';
240                 log_generic_event('s', pcb->pid, pcb->priority, pcb->cmd_str);
241                 delete_process_from_all_queues_except_current(pcb);
242                 pcb->process_status = 21; // STOPPED_BY_SIG
243             }
244             pcb->signals[0] = false;
245             break;
246         case 1: // P_SIGCONT
247             if (pcb->process_state == 'S') { // Only continue if stopped
248                 if (pcb->is_sleeping) {
249                     pcb->process_state = 'B';
250                     delete_process_from_all_queues_except_current(pcb);
251                     put_pcb_into_correct_queue(pcb);
252                 } else {
253                     pcb->process_state = 'R';
254                     delete_process_from_all_queues_except_current(pcb);
255                     put_pcb_into_correct_queue(pcb);
256                 }
257                 log_generic_event('c', pcb->pid, pcb->priority, pcb->cmd_str);
258                 pcb->process_status = 23; // CONT_BY_SIG
259             }
260             pcb->signals[1] = false;
261             break;
262         case 2: // P_SIGTERM
263             if (pcb->process_state != 'Z') { // Don't terminate if already zombie
264                 pcb->process_state = 'Z';
265                 pcb->process_status = 22; // TERM_BY_SIG
266                 log_generic_event('Z', pcb->pid, pcb->priority, pcb->cmd_str);
267                 delete_process_from_all_queues_except_current(pcb);
268                 put_pcb_into_correct_queue(pcb);
269                 pcb->process_status = 22; // TERM_BY_SIG
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 destructors 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_pcb = vec_new(0, free_pcb);
61 }
```

4.15.1.13 put_pcb_into_correct_queue()

```
void put_pcb_into_correct_queue (
    pcb_t * pcb )
```

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 internal 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) {
141             vec_push_back(&one_priority_queue, pcb);
142         } else if (pcb->priority == 2) {
143             vec_push_back(&two_priority_queue, pcb);
144         }
145     } else if (pcb->process_state == 'Z') {
146         vec_push_back(&zombie_queue, pcb);
147     } else if (pcb->process_state == 'B' || pcb->is_sleeping) {
148         vec_push_back(&sleep_blocked_queue, pcb);
149     }
150 }
```

4.15.1.14 s_shutdown_pennos()

```
void s_shutdown_pennos (
    void )
```

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);
306     pthread_sigmask(SIG_UNBLOCK, &alarm_set, NULL);
307
308     struct itimerval it;
309     it.it_interval = (struct timeval){.tv_usec = hundred_millisec};
310     it.it_value = it.it_interval;
311     setitimer(ITIMER_REAL, &it, NULL);
312
313     while (!scheduling_done) {
314         // handle signals for the currently running process
315         if (current_running_pcb != NULL) {
316             for (int i = 0; i < 3; i++) {
317                 if (current_running_pcb->signals[i]) {
318                     handle_signal(current_running_pcb, i);
319                     // If process was terminated, don't continue scheduling it
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)
329         for (int i = 0; i < vec_len(&current_pcb); i++) {
330             pcb_t* curr_pcb = vec_get(&current_pcb, i);
331             for (int j = 0; j < 3; j++) {
332                 if (curr_pcb->signals[j]) {
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++) {
340             pcb_t* blocked_proc = vec_get(&sleep_blocked_queue, i);
341             bool make_runnable = false;
342             if (blocked_proc->is_sleeping &&
343                 blocked_proc->time_to_wake <= tick_counter) {
344                 blocked_proc->is_sleeping = false;
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 &&
349                 blocked_proc->signals[2]) { // P_SIGTERM received
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);
355                 put_pcb_into_correct_queue(blocked_proc);
356                 log_generic_event('Z', blocked_proc->pid, blocked_proc->priority,
357                                 blocked_proc->cmd_str);

```

```

358     i--;
359 } else if (child_in_zombie_queue(blocked_proc)) {
360     make_runnable = true;
361 } else if (child_with_changed_process_status(blocked_proc)) {
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);
368     delete_process_from_all_queues_except_current(blocked_proc);
369     put_pcb_into_correct_queue(blocked_proc);
370     log_generic_event('U', blocked_proc->pid, blocked_proc->priority,
371                     blocked_proc->cmd_str);
372     i--;
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);
392 if (spthread_suspend(current_running_pcb->thread_handle) != 0 &&
393     errno != EINTR) {
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_pcb

```
Vec current_pcb
```

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:

```
= {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.

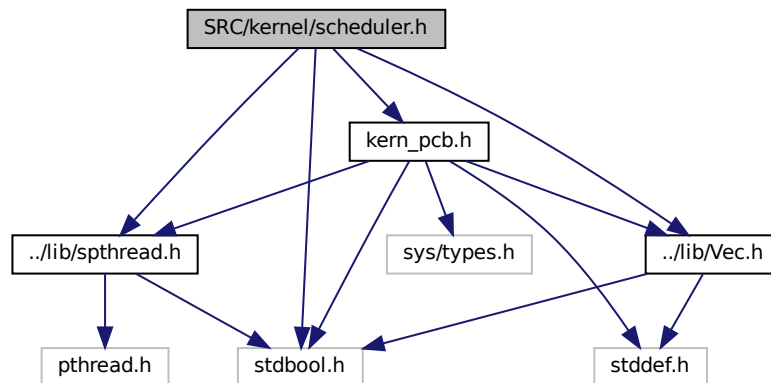
4.15.2.11 zombie_queue

`Vec` zombie_queue

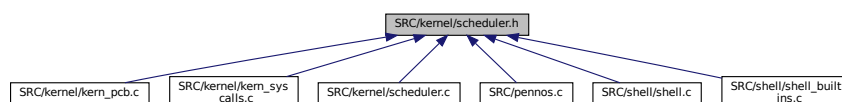
Definition at line 27 of file scheduler.c.

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 probabilities. 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 internal 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 implementation 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 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

<i>signum</i>	The signal number (unused in this implementation).
---------------	--

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()

```
bool child_in_zombie_queue (
    pcb_t * parent )
```

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

<i>parent</i>	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()

```
bool child_with_changed_process_status (
    pcb_t * parent )
```

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

<i>parent</i>	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.

```

215                                     {
216     for (int i = 0; i < vec_len(&current_pcb); i++) {
217         pcb_t* child = vec_get(&current_pcb, i);
218         if (child->par_pid == parent->pid && child->process_status != 0) {
219             return true;
220         }
221     }
222     return false;
223 }
```

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

<i>pcb</i>	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_pcb);
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

<i>pcb</i>	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`.

```

168
169     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 }
```

4.16.1.6 delete_process_from_particular_queue()

```

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 implementation 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 implementation 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++) {
157         pcb_t* curr_pcb = vec_get(queue, i);
158         if (curr_pcb->pid == pcb->pid) {
159             vec_erase_no_deletor(queue, i);
160             return;
161         }
162     }
163 }
```

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`.

```

66     {
67         vec_destroy(&zero_priority_queue);
68         vec_destroy(&one_priority_queue);
69         vec_destroy(&two_priority_queue);
70         vec_destroy(&zombie_queue);
71         vec_destroy(&sleep_blocked_queue);
72         vec_destroy(&current_pcb);
73 }
```

4.16.1.8 generate_next_priority()

```
int generate_next_priority ( )
```

Deterministically chooses an integer from 0, 1, 2 at the prescribed probabilities. 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 is 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 probabilities. 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.

```
83     {
84     // 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     }
89
90     int priorities_attempted = 0;
91     while (priorities_attempted < 19) {
92         int curr_pri = det_priorities_arr[curr_priority_arr_index];
93         curr_priority_arr_index = (curr_priority_arr_index + 1) % 19;
94         if (curr_pri == 0 && !vec_is_empty(&zero_priority_queue)) {
95             priorities_attempted++;
96             return 0;
97         } else if (curr_pri == 1 && !vec_is_empty(&one_priority_queue)) {
98             priorities_attempted++;
99             return 1;
100        } else if (curr_pri == 2 && !vec_is_empty(&two_priority_queue)) {
101            priorities_attempted++;
102            return 2;
103        }
104    }
105
106    return -1; // should never reach
107 }
```

4.16.1.9 get_next_pcb()

```
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.

Parameters

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

4.16.1.10 get_pcb_in_queue()

```

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.

Parameters

<i>queue</i>	the queue of pcb_t* ptrs to search
<i>pid</i>	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.

```

187     {
188     for (int i = 0; i < vec_len(queue); i++) {
189         pcb_t* curr_pcb = vec_get(queue, i);
190         if (curr_pcb->pid == pid) {
191             return curr_pcb;
192         }
193     }
194
195     return NULL;
196 }
```

4.16.1.11 handle_signal()

```
void handle_signal (
    pcb_t * pcb,
    int 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

<i>pcb</i>	A pointer to the PCB of the process receiving the signal.
<i>signal</i>	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.

```
235                                     {
236     switch (signal) {
237     case 0: // P_SIGSTOP
238         if (pcb->process_state == 'R' || pcb->process_state == 'B') {
239             pcb->process_state = 'S';
240             log_generic_event('s', pcb->pid, pcb->priority, pcb->cmd_str);
241             delete_process_from_all_queues_except_current(pcb);
242             pcb->process_status = 21; // STOPPED_BY_SIG
243         }
244         pcb->signals[0] = false;
245         break;
246     case 1: // P_SIGCONT
247         if (pcb->process_state == 'S') { // Only continue if stopped
248             if (pcb->is_sleeping) {
249                 pcb->process_state = 'B';
250                 delete_process_from_all_queues_except_current(pcb);
251                 put_pcb_into_correct_queue(pcb);
252             } else {
253                 pcb->process_state = 'R';
254                 delete_process_from_all_queues_except_current(pcb);
255                 put_pcb_into_correct_queue(pcb);
256             }
257             log_generic_event('c', pcb->pid, pcb->priority, pcb->cmd_str);
258             pcb->process_status = 23; // CONT_BY_SIG
259         }
260         pcb->signals[1] = false;
261         break;
262     case 2: // P_SIGTERM
263         if (pcb->process_state != 'Z') { // Don't terminate if already zombie
264             pcb->process_state = 'Z';
265             pcb->process_status = 22; // TERM_BY_SIG
266             log_generic_event('Z', pcb->pid, pcb->priority, pcb->cmd_str);
267             delete_process_from_all_queues_except_current(pcb);
268             put_pcb_into_correct_queue(pcb);
269             pcb->process_status = 22; // TERM_BY_SIG
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 queues. This function should be called before any other scheduler functions are called.

Note

The destructors 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_pcb = vec_new(0, free_pcb);
61 }
```

4.16.1.13 put_pcb_into_correct_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 internal 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 internal 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) {
141             vec_push_back(&one_priority_queue, pcb);
142         } else if (pcb->priority == 2) {
143             vec_push_back(&two_priority_queue, pcb);
144         }
145     } else if (pcb->process_state == 'Z') {
146         vec_push_back(&zombie_queue, pcb);
147     } else if (pcb->process_state == 'B' || pcb->is_sleeping) {
148         vec_push_back(&sleep_blocked_queue, pcb);
149     }
150 }
```

4.16.1.14 s_shutdown_pennos()

```
void s_shutdown_pennos (
    void )
```

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);
306     pthread_sigmask(SIG_UNBLOCK, &alarm_set, NULL);
307
308     struct itimerval it;
309     it.it_interval = (struct timeval){.tv_usec = hundred_millisec};
310     it.it_value = it.it_interval;
311     setitimer(ITIMER_REAL, &it, NULL);
312
313     while (!scheduling_done) {
314         // handle signals for the currently running process
315         if (current_running_pcb != NULL) {
316             for (int i = 0; i < 3; i++) {
317                 if (current_running_pcb->signals[i]) {
318                     handle_signal(current_running_pcb, i);
319                     // If process was terminated, don't continue scheduling it
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)
329         for (int i = 0; i < vec_len(&current_pcb); i++) {
330             pcb_t* curr_pcb = vec_get(&current_pcb, i);
331             for (int j = 0; j < 3; j++) {
332                 if (curr_pcb->signals[j]) {
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++) {
340             pcb_t* blocked_proc = vec_get(&sleep_blocked_queue, i);
341             bool make_runnable = false;
342             if (blocked_proc->is_sleeping &&
343                 blocked_proc->time_to_wake <= tick_counter) {
344                 blocked_proc->is_sleeping = false;
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 &&
349                 blocked_proc->signals[2]) { // P_SIGTERM received
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);
355                 put_pcb_into_correct_queue(blocked_proc);
356                 log_generic_event('Z', blocked_proc->pid, blocked_proc->priority,
357                                 blocked_proc->cmd_str);

```

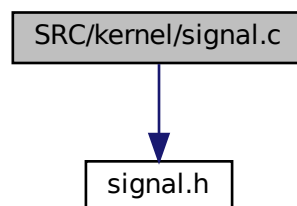
```

358     i--;
359 } else if (child_in_zombie_queue(blocked_proc)) {
360     make_runnable = true;
361 } else if (child_with_changed_process_status(blocked_proc)) {
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);
368     delete_process_from_all_queues_except_current(blocked_proc);
369     put_pcb_into_correct_queue(blocked_proc);
370     log_generic_event('U', blocked_proc->pid, blocked_proc->priority,
371                     blocked_proc->cmd_str);
372     i--;
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);
392 if (spthread_suspend(current_running_pcb->thread_handle) != 0 &&
393     errno != EINTR) {
394     perror("spthread_suspend failed in scheduler");
395 }
396 put_pcb_into_correct_queue(current_running_pcb);
397 }
398 }

```

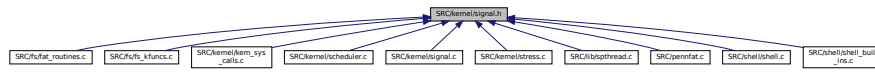
4.17 SRC/kernel/signal.c File Reference

#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

```
#define P_WIFEXITED(  
    status ) ((status) == EXITED_NORMALLY)
```

User-level macros for waitpid status.

Definition at line 22 of file signal.h.

4.18.1.7 P_WIFSIGNALED

```
#define P_WIFSIGNALED(  
    status ) ((status) == TERM_BY_SIG)
```

Definition at line 24 of file signal.h.

4.18.1.8 P_WIFSTOPPED

```
#define P_WIFSTOPPED (
    status ) ((status) == STOPPED_BY_SIG)
```

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

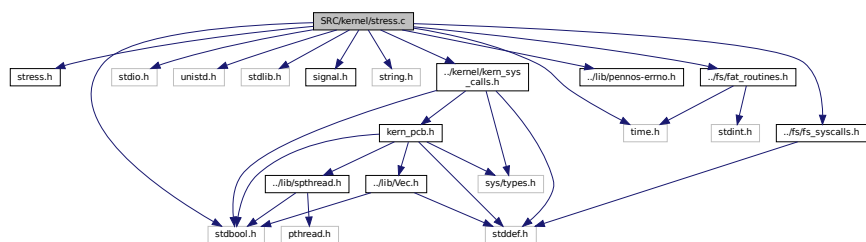
```
#define TERM_BY_SIG 22
```

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 "../lib/pennos-errno.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()

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

Definition at line 233 of file stress.c.

```
233     {
234     // This one only works on a file system big enough to hold 5480 bytes
235     crash_main();
236     s_exit();
237     return NULL;
238 }
```

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.

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

4.19.1.4 recur()

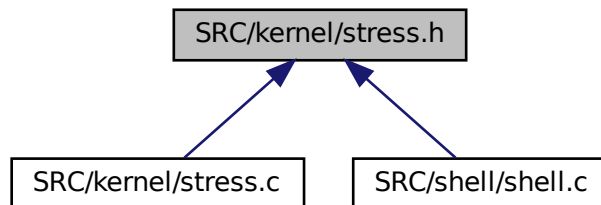
```
void* recur (
    void * arg )
```

Definition at line 227 of file stress.c.

```
227      {
228      spawn_r (NULL);
229      s_exit ();
230      return NULL;
231  }
```

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()

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

Definition at line 233 of file stress.c.

```
233      {
234      // This one only works on a file system big enough to hold 5480 bytes
235      crash_main ();
236      s_exit ();
237      return NULL;
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 )
```

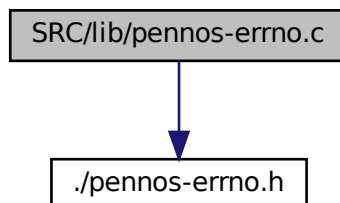
Definition at line 227 of file stress.c.

```
227     {
228     spawn_r(NULL);
229     s_exit();
230     return NULL;
231 }
```

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

```
#include " ./pennos-errno.h"
```

Include dependency graph for pennos-errno.c:



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

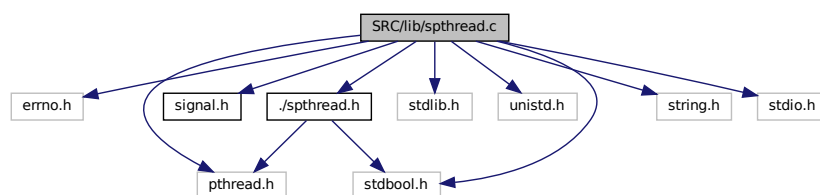
```
int P_ERRNO [extern]
```

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

```
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

```
typedef struct spthread_signal_args_st spthread_signal_args
```

4.23.3 Function Documentation

4.23.3.1 spthread_cancel()

```
int spthread_cancel (  
    spthread_t thread )
```

Definition at line 293 of file `spthread.c`.

```
293 {  
294     return pthread_cancel(thread.thread);  
295 }
```

4.23.3.2 `spthread_continue()`

```
int pthread_continue (
    pthread_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) {
245         // I am already running... so just return 0
246         my_meta->state = SPHTHREAD_RUNNING_STATE;
247         return 0;
248     }
249
250     pthread_signal_args args = (pthread_signal_args){
251         .signal = SPHTHREAD_SIG_CONTINUE,
252         .ack = 0,
253     };
254     pthread_mutex_init(&args.shutup_mutex, NULL);
255
256     int ret = pthread_sigqueue(thread.thread, SIGPTH,
257                               (union sigval){
258                                   .sival_ptr = &args,
259                               });
260     if (ret != 0) {
261         pthread_mutex_destroy(&args.shutup_mutex);
262         // handles the case where the thread is already dead.
263         return ret;
264     }
265
266     // wait for our signal to be ack'd
267
268     // setting up args to nanosleep
269     const struct timespec t = (struct timespec){
270         .tv_nsec = MILLISEC_IN_NANO,
271     };
272
273     pthread_mutex_lock(&args.shutup_mutex);
274     while (args.ack != 1) {
275         // wait for a mili second
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
283         if (thread.meta->state == SPHTHREAD_TERMINATED_STATE) {
284             // child called exit, can break
285             break;
286         }
287     }
288     pthread_mutex_unlock(&args.shutup_mutex);
289     pthread_mutex_destroy(&args.shutup_mutex);
290     return ret;
291 }
```

4.23.3.3 `spthread_create()`

```
int pthread_create (
    pthread_t * thread,
    const pthread_attr_t * attr,
    pthread_fn_t start_routine,
    void * arg )
```

Definition at line 114 of file `spthread.c`.

```
117                                     {
118     pthread_meta_t* child_meta = malloc(sizeof(pthread_meta_t));
119     if (child_meta == NULL) {
120         return EAGAIN;
121     }
122 }
```



```

123  pthread_fwd_args* fwd_args = malloc(sizeof(pthread_fwd_args));
124  if (fwd_args == NULL) {
125      free(child_meta);
126      return EAGAIN;
127  }
128  *fwd_args = (pthread_fwd_args){
129      .actual_routine = start_routine,
130      .actual_arg = arg,
131      .setup_done = false,
132      .child_meta = child_meta,
133  };
134
135  int ret = pthread_mutex_init(&(fwd_args->setup_mutex), NULL);
136  if (ret != 0) {
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) {
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, pthread_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
163  *thread = (pthread_t){
164      .thread = pthread,
165      .meta = child_meta,
166  };
167
168  return result;
169 }

```

4.23.3.4 pthread_disable_interrupts_self()

```
int pthread_disable_interrupts_self ( )
```

Definition at line 326 of file pthread.c.

```

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

```

4.23.3.5 `spthread_enable_interrupts_self()`

```
int pthread_enable_interrupts_self ( )
```

Definition at line 345 of file `spthread.c`.

```
345                                     {
346     sigset_t block_set;
347     int res = sigemptyset(&block_set);
348     if (res != 0) {
349         return res;
350     }
351     res = sigaddset(&block_set, SIGPTHD);
352     if (res != 0) {
353         return res;
354     }
355     res = pthread_sigmask(SIG_UNBLOCK, &block_set, NULL);
356     if (res != 0) {
357         return res;
358     }
359     return 0;
360 }
```

4.23.3.6 `spthread_equal()`

```
bool pthread_equal (
    pthread_t first,
    pthread_t second )
```

Definition at line 322 of file `spthread.c`.

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

4.23.3.7 `spthread_exit()`

```
void pthread_exit (
    void * status )
```

Definition at line 315 of file `spthread.c`.

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

4.23.3.8 `spthread_join()`

```
int pthread_join (
    pthread_t thread,
    void ** retval )
```

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 pthread_self()

```
bool pthread_self (
    pthread_t * thread )
```

Definition at line 297 of file pthread.c.

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

4.23.3.10 pthread_suspend()

```
int pthread_suspend (
    pthread_t thread )
```

Definition at line 171 of file pthread.c.

```
171                                     {
172     pthread_t pself = pthread_self();
173
174     if (pthread_equal(pself, thread.thread) != 0) {
175         return pthread_suspend_self();
176     }
177
178     pthread_signal_args args = (pthread_signal_args){
179         .signal = SPTHREAD_SIG_SUSPEND,
180         .ack = 0,
181     };
182     pthread_mutex_init(&args.shutup_mutex, NULL);
183
184     int ret = pthread_sigqueue(thread.thread, SIGPTH,
185                               (union sigval){
186                                   .sival_ptr = &args,
187                               });
188     if (ret != 0) {
189         pthread_mutex_destroy(&args.shutup_mutex);
190         // handles the case where the thread is already dead.
191         return ret;
192     }
193
194     // wait for our signal to be ack'd
195
196     // setting up args to nanosleep
197     const struct timespec t = (struct timespec){
198         .tv_nsec = MILLISEC_IN_NANO,
199     };
200
201     nanosleep(&t, NULL);
202
203     pthread_mutex_lock(&args.shutup_mutex);
204     while (args.ack != 1) {
205         // wait for a mili second
206         pthread_mutex_unlock(&args.shutup_mutex);
207
208         nanosleep(&t, NULL);
209
210         // fprintf(stderr, "susp checking...\n");
211         pthread_mutex_lock(&args.shutup_mutex);
212
213         if (thread.meta->state == SPTHREAD_TERMINATED_STATE) {
214             // child called exit, can break
215             break;
216         }
217     }
218
219     pthread_mutex_unlock(&args.shutup_mutex);
220
221     pthread_mutex_destroy(&args.shutup_mutex);
222     return ret;
223 }
```

4.23.3.11 spthread_suspend_self()

```
int spthread_suspend_self ( )
```

Definition at line 225 of file spthread.c.

```

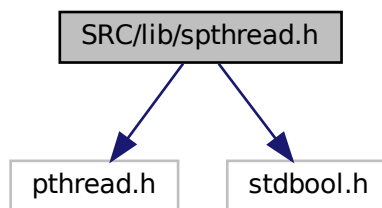
225     {
226     spthread_t self;
227     bool am_sp = spthread_self(&self);
228     if (!am_sp) {
229         return ESRCH;
230     }
231
232     my_meta->state = SPTHREAD_SUSPENDED_STATE;
233
234     do {
235         sigsuspend(&my_meta->suspend_set);
236     } while (my_meta->state == SPTHREAD_SUSPENDED_STATE);
237
238     return 0;
239 }
```

4.24 SRC/lib/spthread.h File Reference

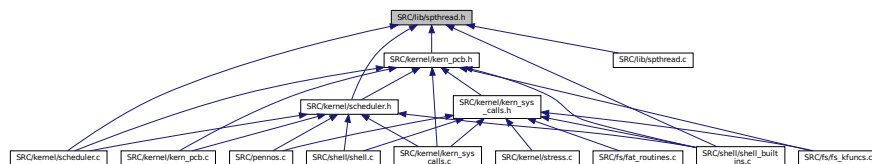
```
#include <pthread.h>
```

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

Include dependency graph for spthread.h:



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



Classes

- struct [spthread_st](#)

Macros

- `#define SIGPTHD SIGUSR1`

Typedefs

- typedef struct `spthread_meta_st` `spthread_meta_t`
- typedef struct `spthread_st` `spthread_t`

Functions

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

4.24.1 Macro Definition Documentation

4.24.1.1 SIGPTHD

```
#define SIGPTHD SIGUSR1
```

Definition at line 19 of file `spthread.h`.

4.24.2 Typedef Documentation

4.24.2.1 `spthread_meta_t`

```
typedef struct spthread_meta_st spthread_meta_t
```

Definition at line 1 of file `spthread.h`.

4.24.2.2 spthread_t

```
typedef struct spthread_st spthread_t
```

4.24.3 Function Documentation

4.24.3.1 spthread_cancel()

```
int spthread_cancel (
    spthread_t thread )
```

Definition at line 293 of file spthread.c.

```
293                                     {
294     return pthread_cancel(thread.thread);
295 }
```

4.24.3.2 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) {
245         // I am already running... so just return 0
246         my_meta->state = SPTHREAD_RUNNING_STATE;
247         return 0;
248     }
249
250     spthread_signal_args args = (spthread_signal_args){
251         .signal = SPTHREAD_SIG_CONTINUE,
252         .ack = 0,
253     };
254     pthread_mutex_init(&args.shutup_mutex, NULL);
255
256     int ret = pthread_sigqueue(thread.thread, SIGPTHD,
257                               (union signal){
258                                   .sival_ptr = &args,
259                               });
260     if (ret != 0) {
261         pthread_mutex_destroy(&args.shutup_mutex);
262         // handles the case where the thread is already dead.
263         return ret;
264     }
265
266     // wait for our signal to be ack'd
267
268     // setting up args to nanosleep
269     const struct timespec t = (struct timespec){
270         .tv_nsec = MILLISEC_IN_NANO,
271     };
272
273     pthread_mutex_lock(&args.shutup_mutex);
274     while (args.ack != 1) {
275         // wait for a mili second
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
283     if (thread.meta->state == SPTHREAD_TERMINATED_STATE) {
284         // child called exit, can break
285         break;
286     }
287 }
288 pthread_mutex_unlock(&args.shutup_mutex);
289 pthread_mutex_destroy(&args.shutup_mutex);
290 return ret;
291 }

```

4.24.3.3 pthread_create()

```

int pthread_create (
    pthread_t * thread,
    const pthread_attr_t * attr,
    void (*)(void *) start_routine,
    void * arg )

```

Definition at line 114 of file pthread.c.

```

117     {
118     pthread_meta_t* child_meta = malloc(sizeof(pthread_meta_t));
119     if (child_meta == NULL) {
120         return EAGAIN;
121     }
122
123     pthread_fwd_args* fwd_args = malloc(sizeof(pthread_fwd_args));
124     if (fwd_args == NULL) {
125         free(child_meta);
126         return EAGAIN;
127     }
128     *fwd_args = (pthread_fwd_args){
129         .actual_routine = start_routine,
130         .actual_arg = arg,
131         .setup_done = false,
132         .child_meta = child_meta,
133     };
134
135     int ret = pthread_mutex_init(&(fwd_args->setup_mutex), NULL);
136     if (ret != 0) {
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) {
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, pthread_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
163     *thread = (pthread_t){
164         .thread = pthread,
165         .meta = child_meta,
166     };
167
168     return result;
169 }

```

4.24.3.4 `spthread_disable_interrupts_self()`

```
int pthread_disable_interrupts_self ( )
```

Definition at line 326 of file `spthread.c`.

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

4.24.3.5 `spthread_enable_interrupts_self()`

```
int pthread_enable_interrupts_self ( )
```

Definition at line 345 of file `spthread.c`.

```
345                                     {
346     sigset_t block_set;
347     int res = sigemptyset(&block_set);
348     if (res != 0) {
349         return res;
350     }
351     res = sigaddset(&block_set, SIGPTHD);
352     if (res != 0) {
353         return res;
354     }
355     res = pthread_sigmask(SIG_UNBLOCK, &block_set, NULL);
356     if (res != 0) {
357         return res;
358     }
359     return 0;
360 }
```

4.24.3.6 `spthread_equal()`

```
bool pthread_equal (
    pthread_t first,
    pthread_t second )
```

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 pthread_exit()

```
void pthread_exit (
    void * status )
```

Definition at line 315 of file pthread.c.

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

4.24.3.8 pthread_join()

```
int pthread_join (
    pthread_t thread,
    void ** retval )
```

Definition at line 308 of file pthread.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 }
```

4.24.3.9 pthread_self()

```
bool pthread_self (
    pthread_t * thread )
```

Definition at line 297 of file pthread.c.

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

4.24.3.10 `spthread_suspend()`

```
int pthread_suspend (
    pthread_t thread )
```

Definition at line 171 of file `spthread.c`.

```
171 {
172     pthread_t pself = pthread_self();
173
174     if (pthread_equal(pself, thread.thread) != 0) {
175         return pthread_suspend_self();
176     }
177
178     pthread_signal_args args = (pthread_signal_args){
179         .signal = SPTHREAD_SIG_SUSPEND,
180         .ack = 0,
181     };
182     pthread_mutex_init(&args.shutup_mutex, NULL);
183
184     int ret = pthread_sigqueue(thread.thread, SIGPTH,
185                               (union sigval){
186                                   .sival_ptr = &args,
187                               });
188     if (ret != 0) {
189         pthread_mutex_destroy(&args.shutup_mutex);
190         // handles the case where the thread is already dead.
191         return ret;
192     }
193
194     // wait for our signal to be ack'd
195
196     // setting up args to nanosleep
197     const struct timespec t = (struct timespec){
198         .tv_nsec = MILLISEC_IN_NANO,
199     };
200
201     nanosleep(&t, NULL);
202
203     pthread_mutex_lock(&args.shutup_mutex);
204     while (args.ack != 1) {
205         // wait for a mili second
206         pthread_mutex_unlock(&args.shutup_mutex);
207
208         nanosleep(&t, NULL);
209
210         // fprintf(stderr, "susp checking...\n");
211         pthread_mutex_lock(&args.shutup_mutex);
212
213         if (thread.meta->state == SPTHREAD_TERMINATED_STATE) {
214             // child called exit, can break
215             break;
216         }
217     }
218
219     pthread_mutex_unlock(&args.shutup_mutex);
220
221     pthread_mutex_destroy(&args.shutup_mutex);
222     return ret;
223 }
```

4.24.3.11 `spthread_suspend_self()`

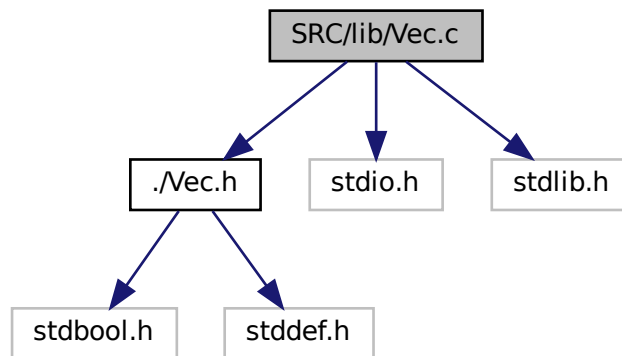
```
int pthread_suspend_self ( )
```

Definition at line 225 of file `spthread.c`.

```
225 {
226     pthread_t self;
227     bool am_sp = pthread_self(&self);
228     if (!am_sp) {
229         return ESRCH;
230     }
231
232     my_meta->state = SPTHREAD_SUSPENDED_STATE;
233
234     do {
235         sigsuspend(&my_meta->suspend_set);
236     } while (my_meta->state == SPTHREAD_SUSPENDED_STATE);
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\(\)](#)

```
void vec_clear (
    Vec * self )
```

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

Parameters

<i>self</i>	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*.

```
34 {
35     if (self->ele_dtor_fn) {
36         for (int i = 0; i < self->length; i++) {
37             self->ele_dtor_fn(self->data[i]);
38         }
39     }
40
41     self->length = 0;
42 }
```

4.25.1.2 vec_destroy()

```
void vec_destroy (
    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

<i>self</i>	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*.

```
17 {
18     if (self->ele_dtor_fn) {
19         for (int i = 0; i < self->length; i++) {
20             self->ele_dtor_fn(self->data[i]);
21         }
22     }
23     free(self->data);
24 }
```

4.25.1.3 vec_erase()

```
void vec_erase (
    Vec * self,
    size_t index )
```

Erases an element at the specified valid location in the container

Parameters

<i>self</i>	a pointer to the vector we want to erase from.
<i>index</i>	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 \geq 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()

```
void vec_erase_no_deletor (
    Vec * self,
    size_t index )
```

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

Parameters

<i>self</i>	a pointer to the vector we want to erase from
<i>index</i>	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()

```
ptr_t vec_get (  
    Vec * self,  
    size_t index )
```

Gets the specified element of the Vec

Parameters

<i>self</i>	a pointer to the vector who's element we want to get.
<i>index</i>	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 \geq 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()

```
void vec_insert (  
    Vec * self,  
    size_t index,  
    ptr_t new_ele )
```

Inserts an element at the specified location in the container

Parameters

<i>self</i>	a pointer to the vector we want to insert into.
<i>index</i>	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.
<i>new_ele</i>	the value we want to insert

Precondition

Assumes self points to a valid vector. If the index is $> \text{self->length}$ 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.

```

91                                     {
92     if (index > self->length) {
93         perror("vec_insert: index > vec length");
94     }
95
96     if (index == self->length) { // Insertion at end = Adding at end
97         vec_push_back(self, new_ele);
98     } else { // Inserting not at the end
99         // Vector is full
100        if (self->length == self->capacity) {
101            vec_resize(self, self->capacity * 2);
102        }
103        // Insertion + Displacement
104        for (unsigned int i = self->length; i > index; i--) {
105            self->data[i] = self->data[i - 1];
106        }
107        self->data[index] = new_ele;
108
109        self->length++;
110    }
111 }
```

4.25.1.7 vec_new()

```

Vec vec_new (
    size_t initial_capacity,
    ptr_dtor_fn ele_dtor_fn )
```

Creates a new empty Vec(tor) with the specified initial_capacity and specified function to clean up elements in the vector.

Parameters

<i>initial_capacity</i>	the initial capacity of the newly created vector, non negative
<i>ele_dtor_fn</i>	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 (
    Vec * self )
```

Removes and destroys the last element of the Vec

Parameters

<i>self</i>	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.

```

113                                     {
114   if (self->length == 0) {
115       return false;
116   }
117   if (self->ele_dtor_fn) {
118       self->ele_dtor_fn(self->data[--self->length]);
119   } else {
120       self->length--;
121   }
122   return true;
123 }
```

4.25.1.9 vec_push_back()

```

void vec_push_back (
    Vec * self,
    ptr_t new_ele )
```

Appends the given element to the end of the Vec

Parameters

<i>self</i>	a pointer to the vector we are pushing onto
<i>new_ele</i>	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) {
127         if (self->capacity == 0) {
128             vec_resize(self, 1);
129         } else {
130             vec_resize(self, self->capacity * 2);
131         }
132     }
133
134     if (self->capacity == self->length) {
135         perror("vec_push_back: resize failed");
136     }
137
138     // The array is 0 indexed
139     self->data[self->length++] = new_ele;
140 }
```

4.25.1.10 vec_resize()

```

void vec_resize (
    Vec * self,
    size_t new_capacity )
```

Resizes the container to a new specified capacity. Does nothing if new_capacity <= self->length

Parameters

<i>self</i>	a pointer to the vector we want to resize.
<i>new_capacity</i>	the new capacity of the vector.

Precondition

Assumes self points to a valid vector.

Postcondition

If a resize takes place, then a reallocation takes place and all elements are copied over. Any pointers to elements prior to this reallocation are invalidated.

The removed elements are destructed (cleaned up).

Definition at line 44 of file Vec.c.

```

44                                     {
45     if (new_capacity * sizeof(void*) < new_capacity) {
46         perror("vec_resize: new capacity too large");
47     }
48     if (new_capacity > self->length) {
49         self->capacity = new_capacity;
50         ptr_t* new_data = malloc(sizeof(void*) * self->capacity);
51
52         // Copy over old elements
53         for (int i = 0; i < self->length; i++) {
54             new_data[i] = self->data[i];
55         }
56         free(self->data);
57         self->data = new_data;
58     }
59 }
61 }
```

4.25.1.11 vec_set()

```

void vec_set (
    Vec * self,
    size_t index,
    ptr_t new_ele )
```

Sets the specified element of the Vec to the specified value

Parameters

<i>self</i>	a pointer to the vector who's element we want to set.
<i>index</i>	the index of the element to set.
<i>new_ele</i>	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 \geq 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 }
```


Functions

- [Vec](#) [vec_new](#) (size_t initial_capacity, [ptr_dtor_fn](#) ele_dtor_fn)
- [ptr_t](#) [vec_get](#) ([Vec](#) *self, size_t index)
- void [vec_set](#) ([Vec](#) *self, size_t index, [ptr_t](#) new_ele)
- void [vec_push_back](#) ([Vec](#) *self, [ptr_t](#) new_ele)
- bool [vec_pop_back](#) ([Vec](#) *self)
- void [vec_insert](#) ([Vec](#) *self, size_t index, [ptr_t](#) new_ele)
- void [vec_erase](#) ([Vec](#) *self, size_t index)
- void [vec_erase_no_deletor](#) ([Vec](#) *self, size_t index)
- void [vec_resize](#) ([Vec](#) *self, size_t new_capacity)
- void [vec_clear](#) ([Vec](#) *self)
- void [vec_destroy](#) ([Vec](#) *self)

4.26.1 Macro Definition Documentation

4.26.1.1 [vec_capacity](#)

```
#define vec_capacity(  
    vec ) ((vec)->capacity)
```

Returns the current capacity of the Vec Written as a function-like macro

Parameters

<i>vec,a</i>	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

<i>vec,a</i>	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 ) ((vec)->length)
```

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

Parameters

<i>vec,a</i>	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 (  
    Vec * self )
```

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

Parameters

<i>self</i>	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*.

```
34 {
35     if (self->ele_dtor_fn) {
36         for (int i = 0; i < self->length; i++) {
37             self->ele_dtor_fn(self->data[i]);
38         }
39     }
40
41     self->length = 0;
42 }
```

4.26.3.2 *vec_destroy()*

```
void vec_destroy (
    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

<i>self</i>	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*.

```
17 {
18     if (self->ele_dtor_fn) {
19         for (int i = 0; i < self->length; i++) {
20             self->ele_dtor_fn(self->data[i]);
21         }
22     }
23     free(self->data);
24 }
```

4.26.3.3 vec_erase()

```
void vec_erase (
    Vec * self,
    size_t index )
```

Erases an element at the specified valid location in the container

Parameters

<i>self</i>	a pointer to the vector we want to erase from.
<i>index</i>	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 \geq 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()

```
void vec_erase_no_deletor (
    Vec * self,
    size_t index )
```

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

Parameters

<i>self</i>	a pointer to the vector we want to erase from
<i>index</i>	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 }

```

4.26.3.5 vec_get()

```

ptr_t vec_get (
    Vec * self,
    size_t index )

```

Gets the specified element of the Vec

Parameters

<i>self</i>	a pointer to the vector who's element we want to get.
<i>index</i>	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 \geq 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()

```

void vec_insert (
    Vec * self,
    size_t index,
    ptr_t new_ele )

```

Inserts an element at the specified location in the container

Parameters

<i>self</i>	a pointer to the vector we want to insert into.
<i>index</i>	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.
<i>new_ele</i>	the value we want to insert

Precondition

Assumes self points to a valid vector. If the index is $> \text{self->length}$ 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.

```

91                                     {
92     if (index > self->length) {
93         perror("vec_insert: index > vec length");
94     }
95
96     if (index == self->length) { // Insertion at end = Adding at end
97         vec_push_back(self, new_ele);
98     } else { // Inserting not at the end
99         // Vector is full
100        if (self->length == self->capacity) {
101            vec_resize(self, self->capacity * 2);
102        }
103        // Insertion + Displacement
104        for (unsigned int i = self->length; i > index; i--) {
105            self->data[i] = self->data[i - 1];
106        }
107        self->data[index] = new_ele;
108
109        self->length++;
110    }
111 }
```

4.26.3.7 vec_new()

```

Vec vec_new (
    size_t initial_capacity,
    ptr_dtor_fn ele_dtor_fn )
```

Creates a new empty Vec(tor) with the specified initial_capacity and specified function to clean up elements in the vector.

Parameters

<i>initial_capacity</i>	the initial capacity of the newly created vector, non negative
<i>ele_dtor_fn</i>	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 (
    Vec * self )
```

Removes and destroys the last element of the Vec

Parameters

<i>self</i>	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.

```

113                                     {
114   if (self->length == 0) {
115       return false;
116   }
117   if (self->ele_dtor_fn) {
118       self->ele_dtor_fn(self->data[--self->length]);
119   } else {
120       self->length--;
121   }
122   return true;
123 }
```

4.26.3.9 vec_push_back()

```

void vec_push_back (
    Vec * self,
    ptr_t new_ele )
```

Appends the given element to the end of the Vec

Parameters

<i>self</i>	a pointer to the vector we are pushing onto
<i>new_ele</i>	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) {
127         if (self->capacity == 0) {
128             vec_resize(self, 1);
129         } else {
130             vec_resize(self, self->capacity * 2);
131         }
132     }
133
134     if (self->capacity == self->length) {
135         perror("vec_push_back: resize failed");
136     }
137
138     // The array is 0 indexed
139     self->data[self->length++] = new_ele;
140 }
```

4.26.3.10 vec_resize()

```

void vec_resize (
    Vec * self,
    size_t new_capacity )
```

Resizes the container to a new specified capacity. Does nothing if new_capacity <= self->length

Parameters

<i>self</i>	a pointer to the vector we want to resize.
<i>new_capacity</i>	the new capacity of the vector.

Precondition

Assumes self points to a valid vector.

Postcondition

If a resize takes place, then a reallocation takes place and all elements are copied over. Any pointers to elements prior to this reallocation are invalidated.

The removed elements are destructed (cleaned up).

Definition at line 44 of file Vec.c.

```

44                                     {
45     if (new_capacity * sizeof(void*) < new_capacity) {
46         perror("vec_resize: new capacity too large");
47     }
48     if (new_capacity > self->length) {
49         self->capacity = new_capacity;
50         ptr_t* new_data = malloc(sizeof(void*) * self->capacity);
51
52         // Copy over old elements
53         for (int i = 0; i < self->length; i++) {
54             new_data[i] = self->data[i];
55         }
56         free(self->data);
57         self->data = new_data;
58     }
59 }
61 }
```

4.26.3.11 vec_set()

```

void vec_set (
    Vec * self,
    size_t index,
    ptr_t new_ele )
```

Sets the specified element of the Vec to the specified value

Parameters

<i>self</i>	a pointer to the vector who's element we want to set.
<i>index</i>	the index of the element to set.
<i>new_ele</i>	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 \geq self->length then this function will call perror

Definition at line 142 of file Vec.c.

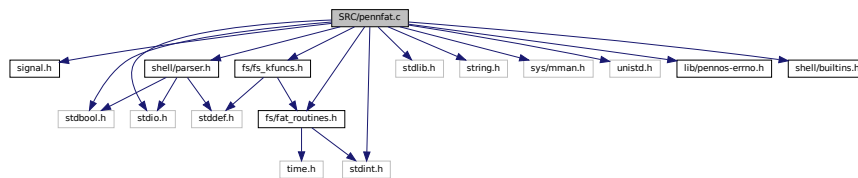
```

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.

```
24      {
25      // register signal handlers
26      struct sigaction sa;
27      sa.sa_handler = signal_handler;
28      sigemptyset(&sa.sa_mask);
29      sa.sa_flags = SA_RESTART;
30
31      // set up handler for SIGINT (ctrl-c)
32      if (sigaction(SIGINT, &sa, NULL) == -1) {
33          P_ERRNO = P_ESIGNAL;
34          u_perror("Error setting up SIGINT handler");
35          return EXIT_FAILURE;
36      }
37
38      // set up handler for SIGTSTP (ctrl-z)
39      if (sigaction(SIGTSTP, &sa, NULL) == -1) {
40          P_ERRNO = P_ESIGNAL;
41          u_perror("Error setting up SIGTSTP handler");
42          return EXIT_FAILURE;
43      }
44
45      char input_buffer[1024];
46
47      while (true) {
48          // print prompt
49          if (k_write(STDOUT_FILENO, PROMPT, strlen(PROMPT)) < 0) {
50              P_ERRNO = P_EWRITE;
51              u_perror("prompt write error");
52              break;
53          }
54
55          // read user input
56          int bytes_read =
57              k_read(STDIN_FILENO, input_buffer, sizeof(input_buffer) - 1);
58
59          // check for EOF (ctrl-D)
60          if (bytes_read <= 0) {
61              k_write(STDOUT_FILENO, "\n", 1);
62              break;
63          }
64
65          // remove trailing newline if present
66          if (bytes_read > 0 && input_buffer[bytes_read - 1] == '\n') {
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);
73          if (parse_result != 0) {
74              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          // extract command and arguments
90          char** args = parsed_command->commands[0];
91
92          // execute command
93          if (strcmp(args[0], "mkfs") == 0) {
94              if (args[1] == NULL || args[2] == NULL || args[3] == NULL) {
95                  P_ERRNO = P_EINVAL;
96                  u_perror("mkfs");
97              } else {
98                  int blocks_in_fat = atoi(args[2]);
99                  int block_size = atoi(args[3]);
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) {
105     if (args[1] == NULL) {
106         P_ERRNO = P_EINVAL;
107         u_perror("mount");
108     } else {
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     }
117 } else if (strcmp(args[0], "ls") == 0) {
118     ls(args);
119 } else if (strcmp(args[0], "touch") == 0) {
120     touch(args);
121 } else if (strcmp(args[0], "cat") == 0) {
122     cat(args);
123 } else if (strcmp(args[0], "chmod") == 0) {
124     chmod(args);
125 } else if (strcmp(args[0], "mv") == 0) {
126     mv(args);
127 } else if (strcmp(args[0], "rm") == 0) {
128     rm(args);
129 } else if (strcmp(args[0], "cp") == 0) {
130     cp(args);
131 } else if (strcmp(args[0], "cmpctdir") == 0) { // extra credit
132     cmpctdir(args);
133 } else {
134     P_ERRNO = P_ECOMMAND;
135     u_perror("shell");
136 }
137
138 free(parsed_command);
139 }
140 return EXIT_SUCCESS;
141 }

```

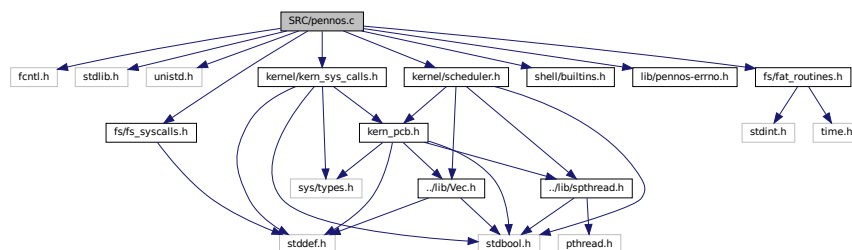
4.28 SRC/pennos.c File Reference

```

#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                                     {
15     // mount the filesystem
16     if (argc < 2) {
17         P_ERRNO = P_NEEDF;
18         u_perror("need a pennfat file to mount");
19         return -1;
20     } else {
21         if (mount(argv[1]) == -1) {
22             u_perror("mount failed");
23             return -1;
24         }
25     }
26
27     // get the log fd
28     if (argc >= 3) {
29         log_fd = open(argv[2], O_RDWR | O_CREAT | O_TRUNC, 0644);
30     } else {
31         log_fd = open("log/log", O_RDWR | O_CREAT | O_TRUNC, 0644);
32     }
33
34     // initialize scheduler architecture and init process
35     initialize_scheduler_queues();
36
37     pid_t init_pid = s_spawn_init();
38     if (init_pid == -1) {
39         P_ERRNO = P_INITFAIL;
40         u_perror("init spawn failed");
41         return -1;
42     }
43
44     scheduler();
45
46     // cleanup
47     s_cleanup_init_process();
48     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

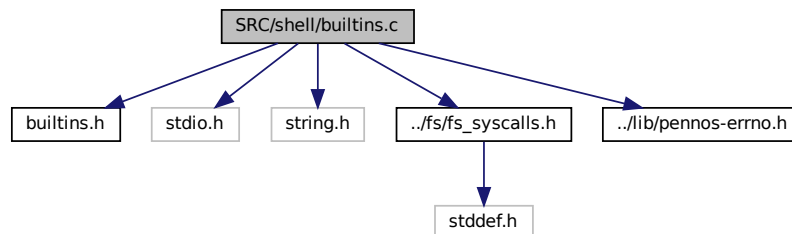
```
int tick_counter [extern]
```

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 (
    const char * msg )
```

Creates a user-level error message similar to perror.

Parameters

<i>msg</i>	A string representing the error message from the shell.
------------	---

Definition at line 15 of file builtins.c.

```

15         {
16     char buffer[256];
17     const char *error_msg;
18
19     switch (P_ERRNO) {
20     case P_ENOENT:
21         error_msg = "file does not exist";
22         break;
23     case P_EBADF:
24         error_msg = "bad file descriptor";
25         break;
26     case P_EPERM:
27         error_msg = "operation not permitted";
28         break;
29     case P_EINVAL:
30         error_msg = "invalid arg";
31         break;
32     case P_EEXIST:
33         error_msg = "file already exists";
34         break;
35     case P_EBUSY:
36         error_msg = "file is busy or open";
37         break;
38     case P_EFULL:
39         error_msg = "no space left on device";
40         break;
41     case P_EINTR:
42         error_msg = "interrupted system call";
43         break;
44     case P_ENULL:
45         error_msg = "NULL returned unexpectedly";
46         break;
47     case P_EUNKNOWN:
48         error_msg = "unknown error";
49         break;
50     case P_EREAD:
51         error_msg = "interrupted read call";
52         break;
53     case P_ELSEEK:
54         error_msg = "interrupted lseek call";
55         break;
56     case P_EMAP:
57         error_msg = "interrupted mmap/munmap call";
58         break;
59     case P_EFUNC:
60         error_msg = "interrupted system call";
61         break;
62     case P_EOPEN:
63         error_msg = "interrupted open call";
64         break;
65     case P_EMALLOC:
66         error_msg = "error when trying to malloc";
67         break;
68     case P_EFS_NOT_MOUNTED:
69         error_msg = "file system not mounted yet";
70         break;
71     case P_ESIGNAL:
72         error_msg = "error with signal handling";
73         break;
74     case P_EWRITE:
75         error_msg = "interrupted write call";
76         break;
77     case P_ECLOSE:
78         error_msg = "interrupted close call";
79         break;
80     case P_EPARSE:
81         error_msg = "error when trying to parse a command";
82         break;
83     case P_ECOMMAND:
84         error_msg = "command not found";
85         break;
86     case P_NEEDF:
87         error_msg = "no file provided to mount";
88         break;
89     case P_EREDIR:
90         error_msg = "input and output cannot be the same when appending";
91         break;
92     default:
93         error_msg = "Unknown error";
94         break;
95     }
96
97     snprintf(buffer, sizeof(buffer), "%s: %s\n", msg, error_msg);
98     if (s_write(STDERR_FILENO, buffer, strlen(buffer)) == -1) {
99         perror("s_write");
100     }

```

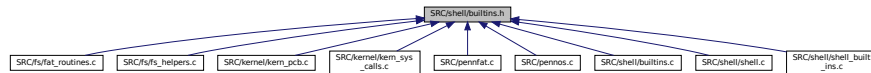
```

101
102     snprintf(buffer, sizeof(buffer), "%s: %s\n", msg, error_msg);
103     s_write(STDERR_FILENO, buffer, strlen(buffer));
104 }

```

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()

```

void u_perror (
    const char * msg )

```

Creates a user-level error message similar to perror.

Parameters

<i>msg</i>	A string representing the error message from the shell.
------------	---

Definition at line 15 of file builtins.c.

```

15     {
16         char buffer[256];
17         const char *error_msg;
18
19         switch (P_ERRNO) {
20             case P_ENOENT:
21                 error_msg = "file does not exist";
22                 break;
23             case P_EBADF:
24                 error_msg = "bad file descriptor";
25                 break;
26             case P_EPERM:
27                 error_msg = "operation not permitted";
28                 break;
29             case P_EINVAL:
30                 error_msg = "invalid arg";
31                 break;
32             case P_EEXIST:
33                 error_msg = "file already exists";
34                 break;

```

```

35     case P_EBUSY:
36         error_msg = "file is busy or open";
37         break;
38     case P_EFULL:
39         error_msg = "no space left on device";
40         break;
41     case P_EINTR:
42         error_msg = "interrupted system call";
43         break;
44     case P_ENULL:
45         error_msg = "NULL returned unexpectedly";
46         break;
47     case P_EUNKNOWN:
48         error_msg = "unknown error";
49         break;
50     case P_EREAD:
51         error_msg = "interrupted read call";
52         break;
53     case P_ELSEEK:
54         error_msg = "interrupted lseek call";
55         break;
56     case P_EMAP:
57         error_msg = "interrupted mmap/munmap call";
58         break;
59     case P_EFUNC:
60         error_msg = "interrupted system call";
61         break;
62     case P_EOPEN:
63         error_msg = "interrupted open call";
64         break;
65     case P_EMALLOC:
66         error_msg = "error when trying to malloc";
67         break;
68     case P_EFS_NOT_MOUNTED:
69         error_msg = "file system not mounted yet";
70         break;
71     case P_ESIGNAL:
72         error_msg = "error with signal handling";
73         break;
74     case P_EWRITE:
75         error_msg = "interrupted write call";
76         break;
77     case P_ECLOSE:
78         error_msg = "interrupted close call";
79         break;
80     case P_EPARSE:
81         error_msg = "error when trying to parse a command";
82         break;
83     case P_ECOMMAND:
84         error_msg = "command not found";
85         break;
86     case P_NEEDF:
87         error_msg = "no file provided to mount";
88         break;
89     case P_EREDIR:
90         error_msg = "input and output cannot be the same when appending";
91         break;
92     default:
93         error_msg = "Unknown error";
94         break;
95 }
96
97 snprintf(buffer, sizeof(buffer), "%s: %s\n", msg, error_msg);
98 if (s_write(STDERR_FILENO, buffer, strlen(buffer)) == -1) {
99     perror("s_write");
100 }
101
102 snprintf(buffer, sizeof(buffer), "%s: %s\n", msg, error_msg);
103 s_write(STDERR_FILENO, buffer, strlen(buffer));
104 }

```

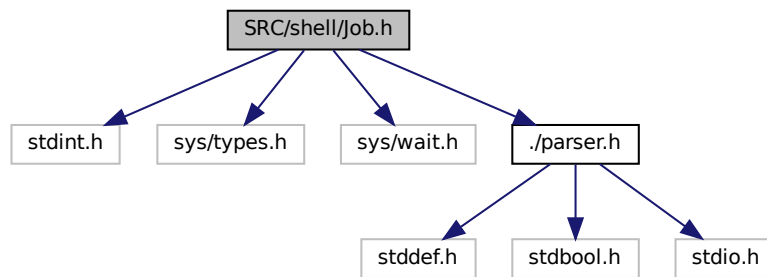
4.31 SRC/shell/Job.h File Reference

```

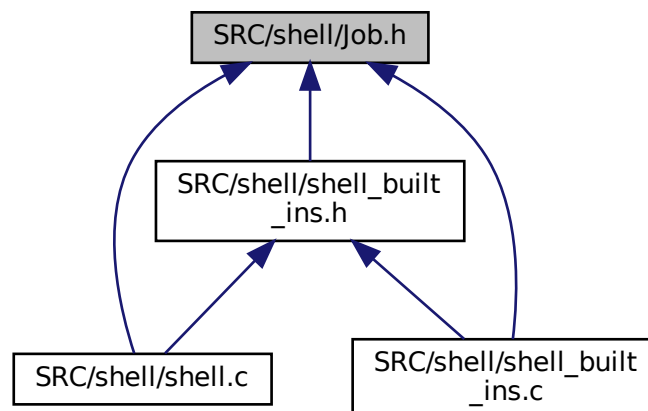
#include <stdint.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "../parser.h"

```

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	

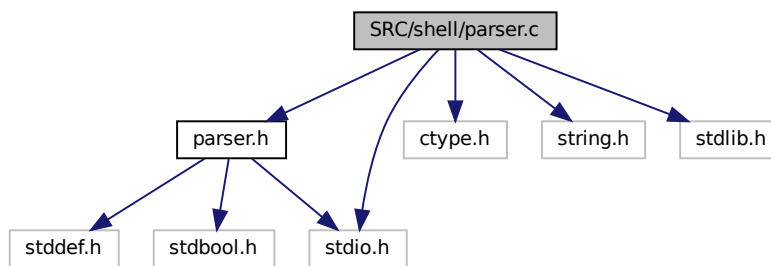
Definition at line 13 of file Job.h.

```
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

```
#define JUMP_OUT(  
    code ) do {ret_code = code; goto PROCESS_ERROR;} while (0)
```

4.32.2 Function Documentation

4.32.2.1 parse_command()

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

Arguments: `cmd_line`: a null-terminated string that is the command line result: a non-null pointer to a `struct parsed_command *`

Return value (int): an error code which can be, 0: parser finished succesfully -1: parser encountered a system call error 1-7: parser specific error, see error type above

This function will parse the given `cmd_line` and store the parsed information into a `struct parsed_command`. The memory needed for the struct will be allocated by this function, and the pointer to the memory will be stored into the given `*result`.

You can directly use the result in system calls. See demo for more information.

If the function returns a successful value (0), a `struct parsed_command` is guaranteed 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`.

```
16
17 #define JUMP_OUT(code) do {ret_code = code; goto PROCESS_ERROR;} while (0)
18
19     int ret_code = -1;
20
21     const char *start = cmd_line;
22     const char *end = cmd_line + strlen(cmd_line);
23
24     for (const char *cur = start; cur < end; ++cur)
25         if (*cur == '#') {
26             // all subsequent characters following '#'
27             // shall be discarded as a comment.
28             end = cur;
29             break;
30         }
31
32     // trimming leading and trailing whitespaces
33     while (start < end && isspace(*start)) ++start;
34     while (start < end && isspace(end[-1])) --end;
35
36     struct parsed_command *pcmd = calloc(1, sizeof(struct parsed_command));
37     if (pcmd == NULL) return -1;
38     if (start == end) goto PROCESS_SUCCESS; // empty line, fast pass
39
40     // If a command is terminated by the control operator ampersand ( '&' ),
41     // the shell shall execute the command in background.
42     if (end[-1] == '&') {
43         pcmd->is_background = true;
44         --end;
```



```

45     }
46
47     // first pass, check token
48     int total_strings = 0; // number of total arguments
49     {
50         bool has_token_last = false, has_file_input = false, has_file_output = false;
51         const char *skipped;
52         for (const char *cur = start; cur < end; skip_space(&cur, end))
53             switch (cur[0]) {
54                 case '&':
55                     JUMP_OUT(UNEXPECTED_AMPERSAND); // does not expect anymore ampersand
56                 case '<':
57                     // if already had pipeline or had file input, error
58                     if (pcmd->num_commands > 0 || has_file_input) JUMP_OUT(UNEXPECTED_FILE_INPUT);
59
60                     ++cur; // skip '<'
61                     skip_space(&cur, end);
62
63                     // test if we indeed have a filename following '<'
64                     skipped = cur;
65                     skip_word(&skipped, end);
66                     if (skipped <= cur) JUMP_OUT(EXPECT_INPUT_FILENAME);
67
68                     // fast-forward to the end of the filename
69                     cur = skipped;
70                     has_file_input = true;
71                     break;
72                 case '>':
73                     // if already had file output, error
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;
85                     skip_word(&skipped, end);
86                     if (skipped <= cur) JUMP_OUT(EXPECT_OUTPUT_FILENAME);
87
88                     // fast-forward to the end of the filename
89                     cur = skipped;
90                     has_file_output = true;
91                     break;
92                 case '|':
93                     // if already had file output but encounter a pipeline, it should
94                     // rather be a file output error instead of a pipeline one.
95                     if (has_file_output) JUMP_OUT(UNEXPECTED_FILE_OUTPUT);
96                     // if no tokens between two pipelines (or before the first one)
97                     // should throw a pipeline error
98                     if (!has_token_last) JUMP_OUT(UNEXPECTED_PIPELINE);
99                     has_token_last = false;
100                     ++pcmd->num_commands;
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) {
110         // if there are no arguments but has ampersand or file input/output
111         // then we have an error
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
119     // (not supporting line continuation)
120     if (!has_token_last) JUMP_OUT(UNEXPECTED_PIPELINE);
121 }
122 ++pcmd->num_commands;
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147     const size_t start_of_array = offsetof(struct parsed_command, commands) + pcmd->num_commands *
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
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
156     char *const new_start = memcpy(new_buf + start_of_str, start, slen);
157
158     // second pass, put stuff in
159     // no need to check for error anymore
160     size_t cur_cmd = 0;
161     char **argv_ptr = (char **) (new_buf + start_of_array);
162
163     pcmd->commands[cur_cmd] = argv_ptr;
164     for (const char *cur = start; cur < end; skip_space(&cur, end)) {
165         switch (cur[0]) {
166             case '<':
167                 ++cur;
168                 skip_space(&cur, end);
169                 // store input file name into 'stdin_file'
170                 pcmd->stdin_file = new_start + (cur - start);
171                 skip_word(&cur, end);
172                 // at end of the input file name
173                 new_start[cur - start] = '\\0';
174                 break;
175             case '>':
176                 if (pcmd->is_file_append) ++cur; // skip another '>'
177                 ++cur;
178                 skip_space(&cur, end);
179                 // store output file name into 'stdout_file'
180                 pcmd->stdout_file = new_start + (cur - start);
181                 skip_word(&cur, end);
182                 // at end of the output file name
183                 new_start[cur - start] = '\\0';
184                 break;
185             case '|':
186                 // null-terminate the current argv
187                 *(argv_ptr++) = NULL;
188                 // store the next argv head
189                 pcmd->commands[++cur_cmd] = argv_ptr;
190                 ++cur;
191                 break;
192             default:
193                 // at start of the argument string
194                 // store it into the arguments array
195                 *(argv_ptr++) = new_start + (cur - start);
196                 skip_word(&cur, end);
197                 // at end of the argument string
198                 new_start[cur - start] = '\\0';
199         }
200     }
201     // null-terminate the last argv
202     *argv_ptr = NULL;
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
219             if (i == 0 && cmd->stdin_file != NULL)
220                 printf("< %s ", cmd->stdin_file);
221
222             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()

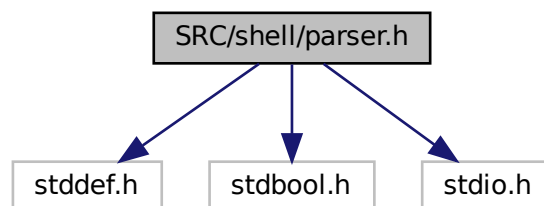
```
void print_parser_errcode (
    FILE * output,
    int err_code )
```

Definition at line 230 of file parser.c.

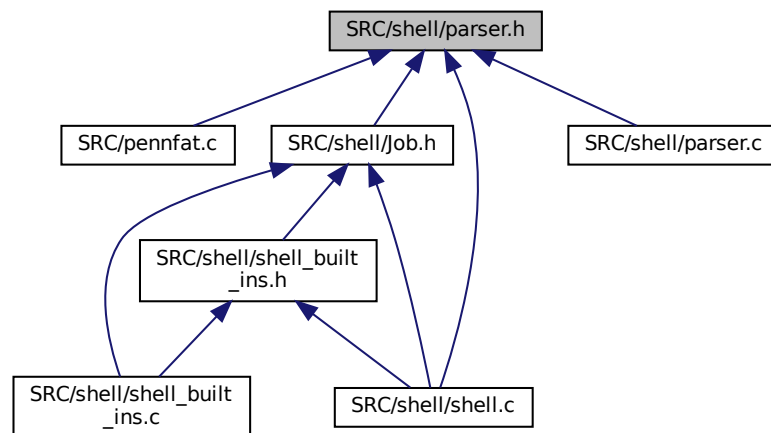
```
230                                     {
231     switch (err_code) {
232     case UNEXPECTED_FILE_INPUT:
233         fprintf(output, "UNEXPECTED INPUT REDIRECTION TO A FILE\n");
234         break;
235     case UNEXPECTED_FILE_OUTPUT:
236         fprintf(output, "UNEXPECTED OUTPUT REDIRECTION TO A FILE\n");
237         break;
238     case UNEXPECTED_PIPELINE:
239         fprintf(output, "UNEXPECTED PIPE\n");
240         break;
241     case UNEXPECTED_AMPERSAND:
242         fprintf(output, "UNEXPECTED AMPERESAND\n");
243         break;
244     case EXPECT_INPUT_FILENAME:
245         fprintf(output, "COULD NOT FINE FILENAME FOR INPUT REDIRECTION \"><\n");
246         break;
247     case EXPECT_OUTPUT_FILENAME:
248         fprintf(output, "COULD NOT FIND FILENAME FOR OUTPUT REDIRECTION \"><\n");
249         break;
250     case EXPECT_COMMANDS:
251         fprintf(output, "COULD NOT FIND ANY COMMANDS OR ARGS\n");
252         break;
253     default:
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()

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

Arguments: cmd_line: a null-terminated string that is the command line result: a non-null pointer to a struct `parsed_command` *

Return value (int): an error code which can be, 0: parser finished succesfully -1: parser encountered a system call error 1-7: parser specific error, see error type above

This function will parse the given `cmd_line` and store the parsed information into a struct `parsed_command`. The memory needed for the struct will be allocated by this function, and the pointer to the memory will be stored into the given `*result`.

You can directly use the result in system calls. See demo for more information.

If the function returns a successful value (0), a struct `parsed_command` is guaranteed 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.

```
16
17 #define JUMP_OUT(code) do {ret_code = code; goto PROCESS_ERROR;} while (0)
18
19     int ret_code = -1;
20
21     const char *start = cmd_line;
22     const char *end = cmd_line + strlen(cmd_line);
23
24     for (const char *cur = start; cur < end; ++cur)
25         if (*cur == '#') {
26             // all subsequent characters following '#'
27             // shall be discarded as a comment.
28             end = cur;
29             break;
30         }
31
32     // trimming leading and trailing whitespaces
33     while (start < end && isspace(*start)) ++start;
34     while (start < end && isspace(end[-1])) --end;
35
36     struct parsed_command *pcmd = calloc(1, sizeof(struct parsed_command));
```

```

37     if (pcmd == NULL) return -1;
38     if (start == end) goto PROCESS_SUCCESS; // empty line, fast pass
39
40     // If a command is terminated by the control operator ampersand ( '&' ),
41     // the shell shall execute the command in background.
42     if (end[-1] == '&') {
43         pcmd->is_background = true;
44         --end;
45     }
46
47     // first pass, check token
48     int total_strings = 0; // number of total arguments
49     {
50         bool has_token_last = false, has_file_input = false, has_file_output = false;
51         const char *skipped;
52         for (const char *cur = start; cur < end; skip_space(&cur, end))
53             switch (cur[0]) {
54                 case '&':
55                     JUMP_OUT(UNEXPECTED_AMPERSAND); // does not expect anymore ampersand
56                 case '<':
57                     // if already had pipeline or had file input, error
58                     if (pcmd->num_commands > 0 || has_file_input) JUMP_OUT(UNEXPECTED_FILE_INPUT);
59
60                     ++cur; // skip '<'
61                     skip_space(&cur, end);
62
63                     // test if we indeed have a filename following '<'
64                     skipped = cur;
65                     skip_word(&skipped, end);
66                     if (skipped <= cur) JUMP_OUT(EXPECT_INPUT_FILENAME);
67
68                     // fast-forward to the end of the filename
69                     cur = skipped;
70                     has_file_input = true;
71                     break;
72                 case '>':
73                     // if already had file output, error
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;
85                     skip_word(&skipped, end);
86                     if (skipped <= cur) JUMP_OUT(EXPECT_OUTPUT_FILENAME);
87
88                     // fast-forward to the end of the filename
89                     cur = skipped;
90                     has_file_output = true;
91                     break;
92                 case '|':
93                     // if already had file output but encounter a pipeline, it should
94                     // rather be a file output error instead of a pipeline one.
95                     if (has_file_output) JUMP_OUT(UNEXPECTED_FILE_OUTPUT);
96                     // if no tokens between two pipelines (or before the first one)
97                     // should throw a pipeline error
98                     if (!has_token_last) JUMP_OUT(UNEXPECTED_PIPELINE);
99                     has_token_last = true;
100                    ++pcmd->num_commands;
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) {
110         // if there are no arguments but has ampersand or file input/output
111         // then we have an error
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
119     // (not supporting line continuation)
120     if (!has_token_last) JUMP_OUT(UNEXPECTED_PIPELINE);
121 }
122 ++pcmd->num_commands;
123

```

```

146
147     const size_t start_of_array = offsetof(struct parsed_command, commands) + pcmd->num_commands *
148     sizeof(char **);
149     const size_t start_of_str = start_of_array + (pcmd->num_commands + total_strings) * sizeof(char *);
150     const size_t slen = end - start;
151
152     char *const new_buf = realloc(pcmd, start_of_str + slen + 1);
153     if (new_buf == NULL) goto PROCESS_ERROR;
154     pcmd = (struct parsed_command *) new_buf;
155
156     // copy string to the new place
157     char *const new_start = memcpy(new_buf + start_of_str, start, slen);
158
159     // second pass, put stuff in
160     // no need to check for error anymore
161     size_t cur_cmd = 0;
162     char **argv_ptr = (char **) (new_buf + start_of_array);
163
164     pcmd->commands[cur_cmd] = argv_ptr;
165     for (const char *cur = start; cur < end; skip_space(&cur, end)) {
166         switch (cur[0]) {
167             case '<':
168                 ++cur;
169                 skip_space(&cur, end);
170                 // store input file name into 'stdin_file'
171                 pcmd->stdin_file = new_start + (cur - start);
172                 skip_word(&cur, end);
173                 // at end of the input file name
174                 new_start[cur - start] = '\\0';
175                 break;
176             case '>':
177                 if (pcmd->is_file_append) ++cur; // skip another '>'
178                 ++cur;
179                 skip_space(&cur, end);
180                 // store output file name into 'stdout_file'
181                 pcmd->stdout_file = new_start + (cur - start);
182                 skip_word(&cur, end);
183                 // at end of the output file name
184                 new_start[cur - start] = '\\0';
185                 break;
186             case '|':
187                 // null-terminate the current argv
188                 *(argv_ptr++) = NULL;
189                 // store the next argv head
190                 pcmd->commands[++cur_cmd] = argv_ptr;
191                 ++cur;
192                 break;
193             default:
194                 // at start of the argument string
195                 // store it into the arguments array
196                 *(argv_ptr++) = new_start + (cur - start);
197                 skip_word(&cur, end);
198                 // at end of the argument string
199                 new_start[cur - start] = '\\0';
200         }
201     }
202     // null-terminate the last argv
203     *argv_ptr = NULL;
204
205     PROCESS_SUCCESS:
206     *result = pcmd;
207     return 0;
208     PROCESS_ERROR:
209     free(pcmd);
210     return ret_code;
211 }

```

4.33.2.2 print_parsed_command()

```

void print_parsed_command (
    const struct parsed_command * 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     }

```



```

219         if (i == 0 && cmd->stdin_file != NULL)
220             printf("< %s ", cmd->stdin_file);
221
222         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     }

```

4.33.2.3 print_parser_errcode()

```

void print_parser_errcode (
    FILE * output,
    int err_code )

```

Definition at line 230 of file parser.c.

```

230     {
231         switch (err_code) {
232             case UNEXPECTED_FILE_INPUT:
233                 fprintf(output, "UNEXPECTED INPUT REDIRECTION TO A FILE\n");
234                 break;
235             case UNEXPECTED_FILE_OUTPUT:
236                 fprintf(output, "UNEXPECTED OUTPUT REDIRECTION TO A FILE\n");
237                 break;
238             case UNEXPECTED_PIPELINE:
239                 fprintf(output, "UNEXPECTED PIPE\n");
240                 break;
241             case UNEXPECTED_AMPERSAND:
242                 fprintf(output, "UNEXPECTED AMPERESAND\n");
243                 break;
244             case EXPECT_INPUT_FILENAME:
245                 fprintf(output, "COULD NOT FINE FILENAME FOR INPUT REDIRECTION \"><\n");
246                 break;
247             case EXPECT_OUTPUT_FILENAME:
248                 fprintf(output, "COULD NOT FIND FILENAME FOR OUTPUT REDIRECTION \"><\n");
249                 break;
250             case EXPECT_COMMANDS:
251                 fprintf(output, "COULD NOT FIND ANY COMMANDS OR ARGS\n");
252                 break;
253             default:
254                 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"

```


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()

```
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.

Parameters

<i>cmd</i>	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) {
```

```

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

```

```

359     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
364     input_fd_script = input_fd;
365     output_fd_script = output_fd;
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;
371     s_waitpid(wait_on, &status, false); // wait for script to finish
372     script_fd = -1; // reset global
373     input_fd_script = STDIN_FILENO;
374     output_fd_script = STDOUT_FILENO;
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
381 return -1; // no matches case
382 }

```

4.34.2.2 fill_buffer_until_full_or_newline()

```

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.

Parameters

<i>fd</i>	the file descriptor to read from, assumed to be open
<i>buffer</i>	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) {
124         int bytes_read = s_read(fd, &currChar, 1);
125         if (bytes_read <= 0 || currChar == '\n') { // EOF or newline cases
126             break;
127         }
128         buffer[i] = currChar;
129         i++;
130     }
131     buffer[i] = '\0'; // Null-terminate the string, replaces \n
132 }

```

4.34.2.3 free_job_ptr()

```

void free_job_ptr (
    void * ptr )

```

Definition at line 102 of file shell.c.

```

102                                     {
103     job* job_ptr = (job*)ptr;
104     free(job_ptr->pids);
105     free(job_ptr);
106 }

```

4.34.2.4 setup_terminal_signal_handlers()

```
void setup_terminal_signal_handlers (
    void )
```

Definition at line 74 of file shell.c.

```
74                                     {
75     struct sigaction sa_int = {0};
76     sa_int.sa_handler = shell_sigint_handler;
77     sigemptyset(&sa_int.sa_mask);
78     sa_int.sa_flags = SA_RESTART;
79     if (sigaction(SIGINT, &sa_int, NULL) == -1) {
80         P_ERRNO = P_ESIGNAL;
81         u_perror("sigaction");
82         exit(EXIT_FAILURE);
83     }
84
85     struct sigaction sa_stp = {0};
86     sa_stp.sa_handler = shell_sigstp_handler;
87     sigemptyset(&sa_stp.sa_mask);
88     sa_stp.sa_flags = SA_RESTART;
89     if (sigaction(SIGTSTP, &sa_stp, NULL) == -1) {
90         P_ERRNO = P_ESIGNAL;
91         u_perror("sigaction");
92         exit(EXIT_FAILURE);
93     }
94 }
```

4.34.2.5 shell()

```
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.

Parameters

<i>input</i>	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
393         while (true) {
394             // poll background jobs
395             int status;
396             pid_t child_pid;
397             while ((child_pid = s_waitpid(-1, &status, true)) > 0) {
398                 // Find which job child_pid belongs to
399                 for (size_t i = 0; i < vec_len(&job_list); i++) {
400                     job* job = vec_get(&job_list, i);
401                     bool in_this_job = false;
402                     for (size_t j = 0; j < job->num_pids; j++) {
403                         if (job->pids[j] == child_pid) {
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                     }
417                 }
418             }
419         }
420     }
```

```

416         if (job->finished_count == job->num_pids) {
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++) {
421                 char** argv = job->cmd->commands[cmdIdx];
422                 int argIdx = 0;
423                 while (argv[argIdx] != NULL) {
424                     snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
425                     s_write(STDOUT_FILENO, buf, strlen(buf));
426                     argIdx++;
427                 }
428             }
429             snprintf(buf, sizeof(buf), "\n");
430             s_write(STDOUT_FILENO, buf, strlen(buf));
431             vec_erase(&job_list, i);
432         }
433     } else if (P_WIFSTOPPED(status) && job->state == RUNNING) {
434         job->state = STOPPED;
435         char buf[128];
436         snprintf(buf, sizeof(buf), "Stopped: ");
437         s_write(STDOUT_FILENO, buf, strlen(buf));
438         for (size_t cmdIdx = 0; cmdIdx < job->cmd->num_commands; cmdIdx++) {
439             char** argv = job->cmd->commands[cmdIdx];
440             int argIdx = 0;
441             while (argv[argIdx] != NULL) {
442                 snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
443                 s_write(STDOUT_FILENO, buf, strlen(buf));
444                 argIdx++;
445             }
446         }
447         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) {
456     u_perror("prompt s_write error");
457     break;
458 }
459
460 // 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) {
464     u_perror("shell read error");
465     break;
466 } else if (user_input == 0) { // EOF case
467     s_shutdown_pennos();
468     break;
469 }
470
471 buffer[user_input] = '\0';
472 if (buffer[user_input - 1] == '\n') {
473     buffer[user_input - 1] = '\0';
474 }
475
476 struct parsed_command* cmd = NULL;
477 int cmd_parse_res = parse_command(buffer, &cmd);
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
485 if (cmd->num_commands == 0) {
486     free(cmd);
487     continue;
488 }
489
490 child_pid = execute_command(cmd);
491 if (child_pid < 0) {
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.
502     job* new_job = malloc(sizeof(job));

```

```

503     if (new_job == NULL) {
504         perror("Error: mallocing new_job failed");
505         free(cmd);
506         continue;
507     }
508     new_job->id = next_job_id++;
509     new_job->pgid = child_pid; // For single commands, child's pid = pgid.
510     new_job->num_pids = 1;
511     new_job->pids = malloc(sizeof(pid_t));
512     if (new_job->pids == NULL) {
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;
519     new_job->state = RUNNING;
520     new_job->cmd = cmd; // Retain command info; do not free here.
521     new_job->finished_count = 0;
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];
526     snprintf(msg, sizeof(msg), "[%lu] %d\n", (unsigned long)new_job->id,
527             child_pid);
528     if (s_write(STDOUT_FILENO, msg, strlen(msg)) == -1) {
529         perror("s_write error");
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)
539         job* new_job = malloc(sizeof(job));
540         if (new_job == NULL) {
541             perror("Error: mallocing new_job failed");
542             free(cmd);
543             continue;
544         }
545         new_job->id = next_job_id++;
546         new_job->pgid = child_pid; // For single commands, child's pid = pgid.
547         new_job->num_pids = 1;
548         new_job->pids = malloc(sizeof(pid_t));
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];
563         snprintf(buf, sizeof(buf), "Stopped: ");
564         s_write(STDOUT_FILENO, buf, strlen(buf));
565         for (size_t cmdIdx = 0; cmdIdx < new_job->cmd->num_commands; cmdIdx++) {
566             char** argv = new_job->cmd->commands[cmdIdx];
567             int argIdx = 0;
568             while (argv[argIdx] != NULL) {
569                 snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
570                 s_write(STDOUT_FILENO, buf, strlen(buf));
571                 argIdx++;
572             }
573             snprintf(buf, sizeof(buf), "\n");
574             s_write(STDOUT_FILENO, buf, strlen(buf));
575         }
576     }
577     current_fg_pid = 2;
578
579     // Free cmd memory for foreground commands.
580     // free(cmd); // TODO --> check if this is already freed, it may be
581 }
582 }
583 }
584
585 vec_destroy(&job_list);
586 s_exit();
587 return 0;
588 }

```


4.34.2.6 shell_sigint_handler()

```
void shell_sigint_handler (
    int sig )
```

Definition at line 49 of file shell.c.

```
49      {
50      // If there's a foreground process, forward SIGINT (terminate) to it, as long
51      // as it's not the shell current_fg_pid will also never be 1 (INIT)
52      if (current_fg_pid != 2) {
53          s_kill(current_fg_pid, 2); // P_SIGTERM
54      }
55
56      if (s_write(STDOUT_FILENO, "\n", 1) == -1) {
57          u_perror("s_write error");
58      }
59 }
```

4.34.2.7 shell_sigstp_handler()

```
void shell_sigstp_handler (
    int sig )
```

Definition at line 62 of file shell.c.

```
62      {
63      // 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()

```
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.

Parameters

<i>cmd</i>	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                                     {
144 // check for independently scheduled processes
145 if (strcmp(cmd->commands[0][0], "cat") == 0) {
146     return s_spawn(u_cat, cmd->commands[0], input_fd_script, output_fd_script);
147 } else if (strcmp(cmd->commands[0][0], "sleep") == 0) {
148     return s_spawn(u_sleep, cmd->commands[0], input_fd_script,
149                 output_fd_script);
150 } else if (strcmp(cmd->commands[0][0], "busy") == 0) {
151     return s_spawn(u_busy, cmd->commands[0], input_fd_script, output_fd_script);
152 } else if (strcmp(cmd->commands[0][0], "echo") == 0) {
153     return s_spawn(u_echo, cmd->commands[0], input_fd_script, output_fd_script);
154 } else if (strcmp(cmd->commands[0][0], "ls") == 0) {
155     return s_spawn(u_ls, cmd->commands[0], input_fd_script, output_fd_script);
156 } else if (strcmp(cmd->commands[0][0], "touch") == 0) {
157     return s_spawn(u_touch, cmd->commands[0], input_fd_script,
158                 output_fd_script);
159 } else if (strcmp(cmd->commands[0][0], "mv") == 0) {
160     return s_spawn(u_mv, cmd->commands[0], input_fd_script, output_fd_script);
161 } else if (strcmp(cmd->commands[0][0], "cp") == 0) {
162     return s_spawn(u_cp, cmd->commands[0], input_fd_script, output_fd_script);
163 } else if (strcmp(cmd->commands[0][0], "rm") == 0) {
164     return s_spawn(u_rm, cmd->commands[0], input_fd_script, output_fd_script);
165 } else if (strcmp(cmd->commands[0][0], "chmod") == 0) {
166     return s_spawn(u_chmod, cmd->commands[0], input_fd_script,
167                 output_fd_script);
168 } else if (strcmp(cmd->commands[0][0], "ps") == 0) {
169     return s_spawn(u_ps, cmd->commands[0], input_fd_script, output_fd_script);
170 } else if (strcmp(cmd->commands[0][0], "kill") == 0) {
171     return s_spawn(u_kill, cmd->commands[0], input_fd_script, output_fd_script);
172 } else if (strcmp(cmd->commands[0][0], "zombify") == 0) {
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);
178 } else if (strcmp(cmd->commands[0][0], "hang") == 0) {
179     return s_spawn(hang, cmd->commands[0], input_fd_script, output_fd_script);
180 } else if (strcmp(cmd->commands[0][0], "nohang") == 0) {
181     return s_spawn(nohang, cmd->commands[0], input_fd_script, output_fd_script);
182 } else if (strcmp(cmd->commands[0][0], "recur") == 0) {
183     return s_spawn(recur, cmd->commands[0], input_fd_script, output_fd_script);
184 } else if (strcmp(cmd->commands[0][0], "crash") == 0) {
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;
192 } else if (strcmp(cmd->commands[0][0], "nice_pid") == 0) {
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;
198 } else if (strcmp(cmd->commands[0][0], "bg") == 0) {
199     u_bg(cmd->commands[0]);
200     return 0;
201 } else if (strcmp(cmd->commands[0][0], "fg") == 0) {
202     u_fg(cmd->commands[0]);
203     return 0;
204 } else if (strcmp(cmd->commands[0][0], "jobs") == 0) {
205     u_jobs(cmd->commands[0]);
206     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()

```

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.

Parameters

<i>arg</i>	standard {function name, NULL} args
------------	-------------------------------------

Returns

NULL

Definition at line 224 of file shell.c.

```

224                                     {
225     // read the script line by line, parse each line, and execute the command
226     while (true) {
227         char buffer[MAX_LINE_BUFFER_SIZE];
228         fill_buffer_until_full_or_newline(script_fd, buffer);
229         if (buffer[0] == '\0') {
230             break; // EOF case
231         }
232
233         // parse the command
234         struct parsed_command* cmd = NULL;
235         int parse_result = parse_command(buffer, &cmd);
236         if (parse_result != 0 || cmd == NULL) {
237             P_ERRNO = P_EPARSE;
238             u_perror("parse_command");
239             free(cmd);
240         }
241
242         // execute the command
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;
246             s_waitpid(child_pid, &status, false);
247         } else if (child_pid < 0) { // nothing spawning so safe to free cmd
248             free(cmd);
249         }
250     }
251
252     s_exit(); // exit the script
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.

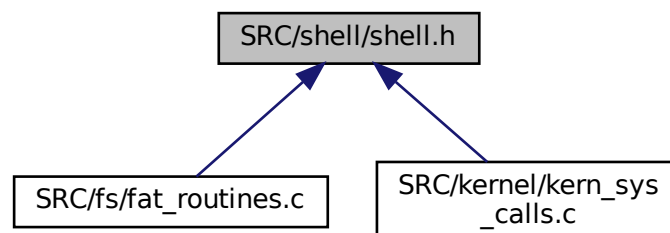
4.34.3.7 script_fd

```
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()`

```
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.

Parameters

<i>input</i>	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
393         while (true) {
394             // poll background jobs
395             int status;
396             pid_t child_pid;
397             while ((child_pid = s_waitpid(-1, &status, true)) > 0) {
398                 // Find which job child_pid belongs to
399                 for (size_t i = 0; i < vec_len(&job_list); i++) {
400                     job* job = vec_get(&job_list, i);
401                     bool in_this_job = false;
402                     for (size_t j = 0; j < job->num_pids; j++) {
403                         if (job->pids[j] == child_pid) {
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) {
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++) {
421                                 char** argv = job->cmd->commands[cmdIdx];
422                                 int argIdx = 0;
423                                 while (argv[argIdx] != NULL) {
```

```

424         snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
425         s_write(STDOUT_FILENO, buf, strlen(buf));
426         argIdx++;
427     }
428 }
429 snprintf(buf, sizeof(buf), "\n");
430 s_write(STDOUT_FILENO, buf, strlen(buf));
431 vec_erase(&job_list, i);
432 }
433 } else if (P_WIFSTOPPED(status) && job->state == RUNNING) {
434     job->state = STOPPED;
435     char buf[128];
436     snprintf(buf, sizeof(buf), "Stopped: ");
437     s_write(STDOUT_FILENO, buf, strlen(buf));
438     for (size_t cmdIdx = 0; cmdIdx < job->cmd->num_commands; cmdIdx++) {
439         char** argv = job->cmd->commands[cmdIdx];
440         int argIdx = 0;
441         while (argv[argIdx] != NULL) {
442             snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
443             s_write(STDOUT_FILENO, buf, strlen(buf));
444             argIdx++;
445         }
446     }
447     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) {
456     u_perror("prompt s_write error");
457     break;
458 }
459
460 // 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) {
464     u_perror("shell read error");
465     break;
466 } else if (user_input == 0) { // EOF case
467     s_shutdown_pennos();
468     break;
469 }
470
471 buffer[user_input] = '\0';
472 if (buffer[user_input - 1] == '\n') {
473     buffer[user_input - 1] = '\0';
474 }
475
476 struct parsed_command* cmd = NULL;
477 int cmd_parse_res = parse_command(buffer, &cmd);
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
485 if (cmd->num_commands == 0) {
486     free(cmd);
487     continue;
488 }
489
490 child_pid = execute_command(cmd);
491 if (child_pid < 0) {
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.
502     job* new_job = malloc(sizeof(job));
503     if (new_job == NULL) {
504         perror("Error: mallocing new_job failed");
505         free(cmd);
506         continue;
507     }
508     new_job->id = next_job_id++;
509     new_job->pgid = child_pid; // For single commands, child's pid = pgid.
510     new_job->num_pids = 1;

```

```

511     new_job->pids = malloc(sizeof(pid_t));
512     if (new_job->pids == NULL) {
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;
519     new_job->state = RUNNING;
520     new_job->cmd = cmd; // Retain command info; do not free here.
521     new_job->finished_count = 0;
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];
526     snprintf(msg, sizeof(msg), "[%lu] %d\n", (unsigned long)new_job->id,
527              child_pid);
528     if (s_write(STDOUT_FILENO, msg, strlen(msg)) == -1) {
529         u_perror("s_write error");
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)
539         job* new_job = malloc(sizeof(job));
540         if (new_job == NULL) {
541             perror("Error: mallocing new_job failed");
542             free(cmd);
543             continue;
544         }
545         new_job->id = next_job_id++;
546         new_job->pgid = child_pid; // For single commands, child's pid = pgid.
547         new_job->num_pids = 1;
548         new_job->pids = malloc(sizeof(pid_t));
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];
563         snprintf(buf, sizeof(buf), "Stopped: ");
564         s_write(STDOUT_FILENO, buf, strlen(buf));
565         for (size_t cmdIdx = 0; cmdIdx < new_job->cmd->num_commands; cmdIdx++) {
566             char** argv = new_job->cmd->commands[cmdIdx];
567             int argIdx = 0;
568             while (argv[argIdx] != NULL) {
569                 snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
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_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 }

```

4.35.2 Variable Documentation

4.35.2.1 is_append

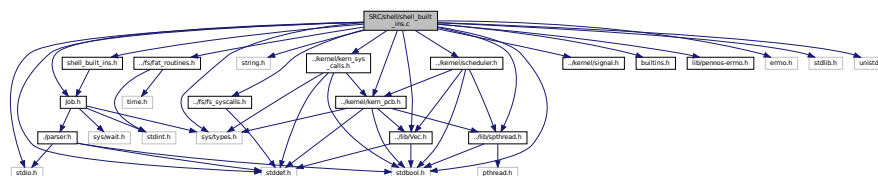
```
int is_append [extern]
```

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:



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 'ls' program built-in that lists files in working directory.
- void * [u_chmod](#) (void *arg)

- Standard 'chmod' program 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 concrete example, if we pass in "cat", we will output "u_cat".*

4.36.1 Function Documentation

4.36.1.1 findJobByIdOrCurrent()

```
job* findJobByIdOrCurrent (
    const char * arg )
```

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.

```
341 {
342     if (vec_len(&job_list) == 0) {
343         return NULL;
344     }
345
346     if (arg != NULL) {
347         // parse numeric
348         char* endPtr = NULL;
349         long val = strtol(arg, &endPtr, 10);
350         if (*endPtr != '\0' || val < 1) {
351             return NULL;
352         }
353         for (size_t i = 0; i < vec_len(&job_list); i++) {
354             job* job_ptr = (job*)vec_get(&job_list, i);
355             if ((jid_t)val == job_ptr->id) {
356                 return job_ptr;
357             }
358         }
359         return NULL;
360     }
361
362     // Look for most recently stopped job first
363     for (size_t i = vec_len(&job_list); i > 0; i--) {
364         job* job_ptr = (job*)vec_get(&job_list, i - 1);
365         if (job_ptr->state == STOPPED) {
366             return job_ptr;
367         }
368     }
369
370     return (job*)vec_get(&job_list, vec_len(&job_list) - 1);
371 }
```

4.36.1.2 orphan_child()

```
void* orphan_child (
    void * arg )
```

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()

```
void* u_bg (
    void * arg )
```

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.

```
377     {
378     char buf[128];
379     char** argv = (char**)arg;
380     const char* jobArg = argv[1]; // NULL if no ID was given
381     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));
385         return NULL;
386     }
387     if (job_ptr->state == STOPPED) {
388         job_ptr->state = RUNNING;
389         snprintf(buf, sizeof(buf), "Running: ");
390         s_write(STDOUT_FILENO, buf, strlen(buf));
391         for (size_t cmdIdx = 0; cmdIdx < job_ptr->cmd->num_commands; cmdIdx++) {
392             char** argv = job_ptr->cmd->commands[cmdIdx];
393             int argIdx = 0;
394             while (argv[argIdx] != NULL) {
395                 snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
396                 s_write(STDOUT_FILENO, buf, strlen(buf));
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;
405     } else if (job_ptr->state == RUNNING) {
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         snprintf(buf, sizeof(buf), "bg: job [%lu] not stopped\n",
412             (unsigned long)job_ptr->id);
413         s_write(STDOUT_FILENO, buf, strlen(buf));
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()

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

The standard 'cat' program built-in.

The usual `cat` program.

Definition at line 42 of file `shell_builtin.c`.

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

4.36.1.6 u_chmod()

```
void* u_chmod (
    void * arg )
```

Standard 'chmod' program 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_builtin.c`.

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

4.36.1.7 u_cp()

```
void* u_cp (
    void * arg )
```

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_builtin.c`.

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

4.36.1.8 u_echo()

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

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()

```
void* u_fg (
    void * arg )
```

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.

```
422      {
423  char buf[128];
424  char** argv = (char**)arg;
425  const char* jobArg = argv[1]; // NULL if no ID was given
426  job* job_ptr = findJobByIdOrCurrent(jobArg);
427  if (!job_ptr) {
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) {
434  snprintf(buf, sizeof(buf), "fg: job [%lu] is already finished\n",
435      (unsigned long)job_ptr->id);
436  s_write(STDOUT_FILENO, buf, strlen(buf));
437  return NULL;
438  }
439
440  if (job_ptr->state == STOPPED) {
441  job_ptr->state = RUNNING;
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];
446  int argIdx = 0;
447  while (argv[argIdx] != NULL) {
448  snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
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 {
458  snprintf(buf, sizeof(buf), "Bringing to foreground: ");
459  s_write(STDOUT_FILENO, buf, strlen(buf));
460  for (size_t cmdIdx = 0; cmdIdx < job_ptr->cmd->num_commands; cmdIdx++) {
461  char** argv = job_ptr->cmd->commands[cmdIdx];
462  int argIdx = 0;
463  while (argv[argIdx] != NULL) {
464  snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
465  s_write(STDOUT_FILENO, buf, strlen(buf));
466  argIdx++;
```

```

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

```

4.36.1.10 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];
513     if (vec_is_empty(&job_list)) {
514         return NULL;
515     }
516
517     for (size_t idx = 0; idx < vec_len(&job_list); idx++) {
518         job* job_ptr = (job*)vec_get(&job_list, idx);
519
520         const char* state = "unknown";
521         if (job_ptr->state == RUNNING) {
522             state = "running";
523         } else if (job_ptr->state == STOPPED) {
524             state = "stopped";
525         } else if (job_ptr->state == FINISHED) {
526             state = "finished";
527         }
528
529         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];
533             int argIdx = 0;
534             while (argv[argIdx] != NULL) {
535                 snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);

```

```

536         s_write(STDOUT_FILENO, buf, strlen(buf));
537         argIdx++;
538     }
539 }
540 snprintf(buf, sizeof(buf), "(%s)\n", state);
541 s_write(STDOUT_FILENO, buf, strlen(buf));
542 }
543 return NULL;
544 }

```

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     int sig = 2;           // Default signal: term (2)
164     int start_index = 1;   // Start after the "kill" command word.
165     char err_buf[128];
166
167     // Check if the first argument specifies a signal
168     if (argv[start_index] && argv[start_index][0] == '-') {
169         if (strcmp(argv[start_index], "-term") == 0) {
170             sig = 2;
171         } else if (strcmp(argv[start_index], "-stop") == 0) {
172             sig = 0;
173         } else if (strcmp(argv[start_index], "-cont") == 0) {
174             sig = 1;
175         } else {
176             // Construct error message
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;
186         long pid_long = strtol(argv[i], &endptr, 10);
187         if (*endptr != '\0' || pid_long <= 0) {
188             snprintf(err_buf, 128, "Invalid PID: %s\n", argv[i]);
189             if (s_write(STDERR_FILENO, err_buf, strlen(err_buf)) == -1) {
190                 u_perror("s_write error");
191             }
192             continue;
193         }
194         pid_t pid = (pid_t)pid_long;
195         if (s_kill(pid, sig) < 0) {
196             snprintf(err_buf, 128, "b_kill error on PID %d\n", pid);
197             if (s_write(STDERR_FILENO, err_buf, strlen(err_buf)) == -1) {
198                 u_perror("s_write error");
199             }
200         }
201     }
202     s_exit();
203     return NULL;
204 }

```

4.36.1.12 u_logout()

```
void* u_logout (
    void * arg )
```

Exits the shell and shuts down PennOS.

Exits the shell and shutdown PennOS.

Definition at line 549 of file shell_builtin.c.

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

4.36.1.13 u_ls()

```
void* u_ls (
    void * arg )
```

Standard 'ls' 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_builtin.c.

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

4.36.1.14 u_man()

```
void* u_man (
    void * arg )
```

Lists all available commands in PennOS in terminal.

Lists all available commands.

Definition at line 300 of file shell_builtin.c.

```
300     {
301         const char* man_string =
302             "cat f1 f2 ...      : concatenates provided files (if none, reads from "
303             "std in), and writes to std out\n"
304             "sleep n              : sleeps for n seconds\n"
305             "busy                  : busy waits indefinitely\n"
306             "echo str              : echoes back the input string str\n"
307             "ls                    : lists all files in the working directory\n"
308             "touch f1 f2 ...        : for each file, creates empty file if it doesn't "
309             "exist yet, otherwise updates its timestamp\n"
310             "mv f1 f2              : renames f1 to f2 (overwrites f2 if it exists)\n"
311             "cp f1 f2              : copies f1 to f2 (overwrites f2 if it exists)\n"
312             "rm f1 f2 ...          : removes the input list of files\n"
313             "chmod +_ f1           : changes f1 permissions to +_ specifications "
314             " (+x, +rw, etc)\n"
315             "ps                  : lists all processes on PennOS, displaying PID, "
316             "PPID, priority, status, and command name\n"
317             "kill (-_) pid1 pid 2 : sends specified signal (term default) to list "
318             "of processes\n"
```



```

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

```

4.36.1.15 u_mv()

```

void* u_mv (
    void * arg )

```

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()

```

void* u_nice (
    void * arg )

```

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     {
256         char* endptr;
257         errno = 0;
258         int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10);
259         if (*endptr != '\0' || errno != 0 || new_priority > 2 ||
260             new_priority < 0) { // error catch
261             return NULL;
262         }
263
264         char* command = ((char**)arg)[2];
265         void* (*ufunc)(void*) = get_associated_ufunc(command);
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
272         if (new_proc_pid != -1) { // non-error case
273             s_nice(new_proc_pid, new_priority);
274         }
275
276         return NULL;
277     }

```

4.36.1.17 u_nice_pid()

```
void* u_nice_pid (
    void * arg )
```

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

4.36.1.18 u_orphanify()

```
void* u_orphanify (
    void * arg )
```

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()

```
void* u_ps (
    void * arg )
```

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()

```
void* u_rm (
    void * arg )
```

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()

```
void* u_sleep (
    void * arg )
```

The standard 'sleep' program built-in.

Sleep for n seconds.

Definition at line 51 of file shell_built_ins.c.

```
51     {
52         char* endptr;
53         errno = 0;
54         if (((char**)arg)[1] == NULL) { // no arg case
55             s_exit();
56             return NULL;
57         }
58         int sleep_secs = (int)strtol(((char**)arg)[1], &endptr, 10);
59         if (*endptr != '\0' || errno != 0 || sleep_secs <= 0) {
60             s_exit();
61             return NULL;
62         }
63
64         int sleep_ticks = sleep_secs * 10;
65         s_sleep(sleep_ticks);
66         s_exit();
67         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()

```
void* zombie_child (
    void * arg )
```

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

```
void*(*) (void*) get_associated_ufunc(char *func) (
    char * func )
```

Helper function to get the associated "u-version" of a function given its standalone version. As a concrete example, if we pass in "cat", we will output "u_cat".

Parameters

<i>func</i>	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.

```

221                                     {
222     if (strcmp(func, "cat") == 0) {
223         return u_cat;
224     } else if (strcmp(func, "sleep") == 0) {
225         return u_sleep;
226     } else if (strcmp(func, "busy") == 0) {
227         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;
232     } else if (strcmp(func, "touch") == 0) {
233         return u_touch;
234     } else if (strcmp(func, "mv") == 0) {
235         return u_mv;
236     } else if (strcmp(func, "cp") == 0) {
237         return u_cp;
238     } else if (strcmp(func, "rm") == 0) {
239         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

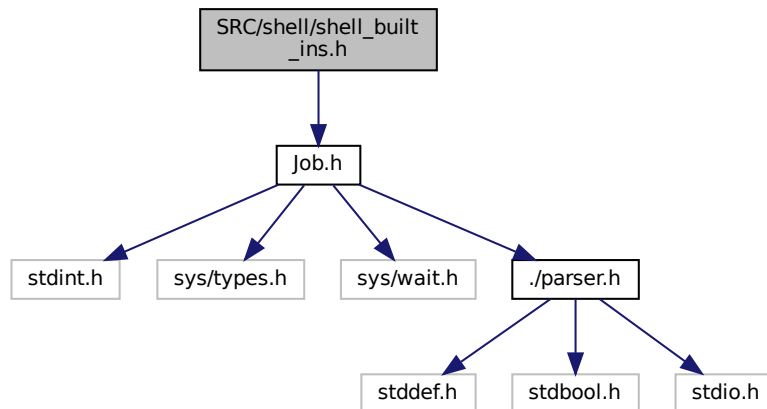
`Vec job_list` [extern]

Definition at line 35 of file shell.c.

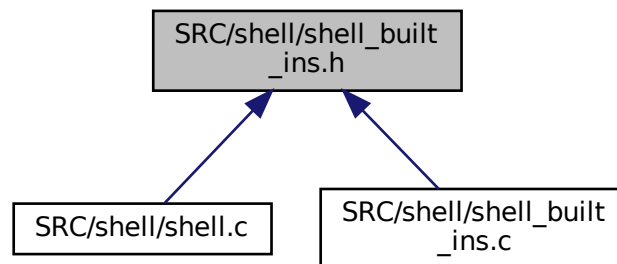
4.37 SRC/shell/shell_built_ins.h File Reference

```
#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 shutdowns 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 findJobByIdOrCurrent()

```
job* findJobByIdOrCurrent (
    const char * arg )
```

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.

```

341                                     {
342     if (vec_len(&job_list) == 0) {
343         return NULL;
344     }
345
346     if (arg != NULL) {
347         // parse numeric
348         char* endPtr = NULL;
349         long val = strtol(arg, &endPtr, 10);
350         if (*endPtr != '\0' || val < 1) {
351             return NULL;
352         }
353         for (size_t i = 0; i < vec_len(&job_list); i++) {
354             job* job_ptr = (job*)vec_get(&job_list, i);
355             if ((jid_t)val == job_ptr->id) {
356                 return job_ptr;
357             }
358         }
359         return NULL;
360     }
361
362     // Look for most recently stopped job first
363     for (size_t i = vec_len(&job_list); i > 0; i--) {
364         job* job_ptr = (job*)vec_get(&job_list, i - 1);
365         if (job_ptr->state == STOPPED) {
366             return job_ptr;
367         }
368     }
369
370     return (job*)vec_get(&job_list, vec_len(&job_list) - 1);
371 }

```

4.37.1.2 u_bg()

```

void* u_bg (
    void * arg )

```

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.

```

377                                     {
378     char buf[128];
379     char** argv = (char**)arg;
380     const char* jobArg = argv[1]; // NULL if no ID was given
381     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));
385         return NULL;
386     }
387     if (job_ptr->state == STOPPED) {
388         job_ptr->state = RUNNING;
389         snprintf(buf, sizeof(buf), "Running: ");
390         s_write(STDOUT_FILENO, buf, strlen(buf));
391         for (size_t cmdIdx = 0; cmdIdx < job_ptr->cmd->num_commands; cmdIdx++) {
392             char** argv = job_ptr->cmd->commands[cmdIdx];
393             int argIdx = 0;
394             while (argv[argIdx] != NULL) {
395                 snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
396                 s_write(STDOUT_FILENO, buf, strlen(buf));
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;
405     } else if (job_ptr->state == RUNNING) {
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     snprintf(buf, sizeof(buf), "bg: job [%lu] not stopped\n",
412             (unsigned long)job_ptr->id);
413     s_write(STDOUT_FILENO, buf, strlen(buf));
414     return NULL;
415 }
416 }
```

4.37.1.3 u_busy()

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

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()

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

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()

```
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.

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()

```
void* u_cp (
    void * arg )
```

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()

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

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()

```
void* u_fg (
    void * arg )
```

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.

```
422      {
423  char buf[128];
424  char** argv = (char**)arg;
425  const char* jobArg = argv[1]; // NULL if no ID was given
426  job* job_ptr = findJobByIdOrCurrent(jobArg);
427  if (!job_ptr) {
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) {
434  snprintf(buf, sizeof(buf), "fg: job [%lu] is already finished\n",
435  (unsigned long)job_ptr->id);
436  s_write(STDOUT_FILENO, buf, strlen(buf));
437  return NULL;
438  }
439
440  if (job_ptr->state == STOPPED) {
441  job_ptr->state = RUNNING;
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];
446  int argIdx = 0;
447  while (argv[argIdx] != NULL) {
448  snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
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 {
458  snprintf(buf, sizeof(buf), "Bringing to foreground: ");
459  s_write(STDOUT_FILENO, buf, strlen(buf));
460  for (size_t cmdIdx = 0; cmdIdx < job_ptr->cmd->num_commands; cmdIdx++) {
```

```

461     char** argv = job_ptr->cmd->commands[cmdIdx];
462     int argIdx = 0;
463     while (argv[argIdx] != NULL) {
464         snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
465         s_write(STDOUT_FILENO, buf, strlen(buf));
466         argIdx++;
467     }
468 }
469 snprintf(buf, sizeof(buf), "\n");
470 }
471
472 current_fg_pid = job_ptr->pids[0];
473
474 while (true) {
475     int status = 0;
476     pid_t wpid = s_waitpid(job_ptr->pgid, &status, false);
477     if (wpid < 0) {
478         if (P_ERRNO == P_EINTR) {
479             continue;
480         }
481         break;
482     }
483     if (P_WIFEXITED(status) || P_WIFSIGNALED(status)) {
484         job_ptr->state = FINISHED;
485         // Remove finished job from list
486         for (size_t i = 0; i < vec_len(&job_list); i++) {
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)) {
495         job_ptr->state = STOPPED;
496         snprintf(buf, sizeof(buf), "Stopped: ");
497         s_write(STDOUT_FILENO, buf, strlen(buf));
498         print_parsed_command(job_ptr->cmd);
499         break;
500     }
501 }
502
503 // back to shell
504 current_fg_pid = 2;
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];
513     if (vec_is_empty(&job_list)) {
514         return NULL;
515     }
516
517     for (size_t idx = 0; idx < vec_len(&job_list); idx++) {
518         job* job_ptr = (job*)vec_get(&job_list, idx);
519
520         const char* state = "unknown";
521         if (job_ptr->state == RUNNING) {
522             state = "running";
523         } else if (job_ptr->state == STOPPED) {
524             state = "stopped";
525         } else if (job_ptr->state == FINISHED) {
526             state = "finished";
527         }
528
529         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];
533         int argIdx = 0;
534         while (argv[argIdx] != NULL) {
535             snprintf(buf, sizeof(buf), "%s ", argv[argIdx]);
536             s_write(STDOUT_FILENO, buf, strlen(buf));
537             argIdx++;
538         }
539     }
540     snprintf(buf, sizeof(buf), "(%s)\n", state);
541     s_write(STDOUT_FILENO, buf, strlen(buf));
542 }
543 return NULL;
544 }

```

4.37.1.10 u_kill()

```

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.

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     {
162     char** argv = (char**)arg;
163     int sig = 2; // Default signal: term (2)
164     int start_index = 1; // Start after the "kill" command word.
165     char err_buf[128];
166
167     // Check if the first argument specifies a signal
168     if (argv[start_index] && argv[start_index][0] == '-') {
169         if (strcmp(argv[start_index], "-term") == 0) {
170             sig = 2;
171         } else if (strcmp(argv[start_index], "-stop") == 0) {
172             sig = 0;
173         } else if (strcmp(argv[start_index], "-cont") == 0) {
174             sig = 1;
175         } else {
176             // Construct error message
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;
186         long pid_long = strtol(argv[i], &endptr, 10);
187         if (*endptr != '\0' || pid_long <= 0) {
188             snprintf(err_buf, 128, "Invalid PID: %s\n", argv[i]);
189             if (s_write(STDERR_FILENO, err_buf, strlen(err_buf)) == -1) {
190                 u_perror("s_write error");
191             }
192             continue;
193         }
194         pid_t pid = (pid_t)pid_long;
195         if (s_kill(pid, sig) < 0) {
196             snprintf(err_buf, 128, "b_kill error on PID %d\n", pid);
197             if (s_write(STDERR_FILENO, err_buf, strlen(err_buf)) == -1) {
198                 u_perror("s_write error");
199             }
200         }
201     }
202     s_exit();
203     return NULL;
204 }

```

4.37.1.11 u_logout()

```
void* u_logout (
    void * arg )
```

Exits the shell and shutdown PennOS.

Example Usage: logout

Exits the shell and shutdown 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()

```
void* u_ls (
    void * arg )
```

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

Example Usage: ls (regular credit) Example Usage: ls ../../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()

```
void* u_man (
    void * arg )
```

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

```

311     "cp f1 f2           : copies f1 to f2 (overwrites f2 if it exists)\n"
312     "rm f1 f2 ...       : removes the input list of files\n"
313     "chmod +_ f1        : changes f1 permissions to +_ specifications "
314     "(+x, +rw, etc)\n"
315     "ps                 : lists all processes on PennOS, displaying PID, "
316     "PPID, priority, status, and command name\n"
317     "kill (___) pid1 pid 2 : sends specified signal (term default) to list "
318     "of processes\n"
319     "nice n command      : spawns a new process for command and sets its "
320     "priority to n\n"
321     "nice_pid n pid      : adjusts the priority level of an existing "
322     "process to n\n"
323     "man                 : lists all available commands in PennOS\n"
324     "bg                  : resumes most recently stopped process in "
325     "background or the one specified by job_id\n"
326     "fg                  : brings most recently stopped or background job "
327     "to foreground or the one specified by job_id\n"
328     "jobs                 : lists all jobs\n"
329     "logout               : exits the shell and shuts down PennOS\n"
330     "zombify              : creates a child process that becomes a zombie\n"
331     "orphanify            : creates a child process that becomes an "
332     "orphan\n";
333
334     s_write(STDOUT_FILENO, man_string, strlen(man_string));
335     return NULL;
336 }

```

4.37.1.14 u_mv()

```

void* u_mv (
    void * arg )

```

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()

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

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

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

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

Definition at line 255 of file `shell_built_ins.c`.

```
255     {
256     char* endptr;
257     errno = 0;
258     int new_priority = (int)strtol(((char**)arg)[1], &endptr, 10);
259     if (*endptr != '\0' || errno != 0 || new_priority > 2 ||
260         new_priority < 0) { // error catch
261         return NULL;
262     }
263
264     char* command = ((char**)arg)[2];
265     void* (*ufunc)(void*) = get_associated_ufunc(command);
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
272     if (new_proc_pid != -1) { // non-error case
273         s_nice(new_proc_pid, new_priority);
274     }
275
276     return NULL;
277 }
```

4.37.1.16 u_nice_pid()

```
void* u_nice_pid (
    void * arg )
```

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


4.37.1.17 u_orphanify()

```
void* u_orphanify (
    void * arg )
```

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.

```
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     }
```

4.37.1.18 u_ps()

```
void* u_ps (
    void * arg )
```

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.

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

4.37.1.19 u_rm()

```
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.)

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;
145     }
```

4.37.1.20 u_sleep()

```
void* u_sleep (
    void * arg )
```

Sleep for *n* seconds.

Note that you'll have to convert the number of seconds to the correct number of ticks.

Example Usage: sleep 10

Sleep for *n* seconds.

Definition at line 51 of file shell_built_ins.c.

```
51     {
52     char* endptr;
53     errno = 0;
54     if (((char**)arg)[1] == NULL) { // no arg case
55         s_exit();
56         return NULL;
57     }
58     int sleep_secs = (int)strtol(((char**)arg)[1], &endptr, 10);
59     if (*endptr != '\0' || errno != 0 || sleep_secs <= 0) {
60         s_exit();
61         return NULL;
62     }
63
64     int sleep_ticks = sleep_secs * 10;
65     s_sleep(sleep_ticks);
66     s_exit();
67     return NULL;
68 }
```

4.37.1.21 u_touch()

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

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

Example Usage: touch f1 f2 f3 f4 f5

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()

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

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.

```
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 }
```

Index

- [_GNU_SOURCE](#)
 - [spthread.c, 195](#)
- [ack](#)
 - [spthread_signal_args_st, 18](#)
- [actual_arg](#)
 - [spthread_fwd_args_st, 16](#)
- [actual_routine](#)
 - [spthread_fwd_args_st, 16](#)
- [add_file_entry](#)
 - [fs_helpers.c, 57](#)
 - [fs_helpers.h, 72](#)
- [alarm_handler](#)
 - [scheduler.c, 163](#)
 - [scheduler.h, 173](#)
- [allocate_block](#)
 - [fs_helpers.c, 59](#)
 - [fs_helpers.h, 74](#)
- [block_size](#)
 - [fs_helpers.c, 70](#)
 - [fs_helpers.h, 87](#)
- [builtins.c](#)
 - [u_perror, 231](#)
- [builtins.h](#)
 - [u_perror, 233](#)
- [capacity](#)
 - [vec_st, 20](#)
- [cat](#)
 - [fat_routines.c, 24](#)
 - [fat_routines.h, 41](#)
- [child_in_zombie_queue](#)
 - [scheduler.c, 163](#)
 - [scheduler.h, 174](#)
- [child_meta](#)
 - [spthread_fwd_args_st, 16](#)
- [child_pcb](#)
 - [pcb_st, 13](#)
- [child_with_changed_process_status](#)
 - [scheduler.c, 164](#)
 - [scheduler.h, 174](#)
- [chmod](#)
 - [fat_routines.c, 28](#)
 - [fat_routines.h, 45](#)
- [cmd](#)
 - [job_st, 9](#)
- [cmd_str](#)
 - [pcb_st, 13](#)
- [cmpctdir](#)
 - [fat_routines.c, 29](#)
 - [fat_routines.h, 46](#)
- [commands](#)
 - [parsed_command, 11](#)
- [compact_directory](#)
 - [fs_helpers.c, 59](#)
 - [fs_helpers.h, 75](#)
- [CONT_BY_SIG](#)
 - [signal.h, 183](#)
- [copy_host_to_pennfat](#)
 - [fs_helpers.c, 62](#)
 - [fs_helpers.h, 78](#)
- [copy_pennfat_to_host](#)
 - [fs_helpers.c, 63](#)
 - [fs_helpers.h, 79](#)
- [copy_source_to_dest](#)
 - [fs_helpers.c, 64](#)
 - [fs_helpers.h, 80](#)
- [cp](#)
 - [fat_routines.c, 29](#)
 - [fat_routines.h, 47](#)
- [crash](#)
 - [stress.c, 186](#)
 - [stress.h, 187](#)
- [create_pcb](#)
 - [kern_pcb.c, 125](#)
 - [kern_pcb.h, 130](#)
- [curr_priority_arr_index](#)
 - [scheduler.c, 170](#)
- [current_fg_pid](#)
 - [fs_kfuncs.c, 100](#)
 - [kern_sys_calls.c, 143](#)
 - [shell.c, 257](#)
 - [shell_built_ins.c, 274](#)
- [current_pcb](#)
 - [kern_pcb.c, 128](#)
 - [kern_sys_calls.c, 143](#)
 - [scheduler.c, 170](#)
- [current_running_pcb](#)
 - [fat_routines.c, 37](#)
 - [fs_kfuncs.c, 100](#)
 - [kern_pcb.c, 128](#)
 - [kern_sys_calls.c, 143](#)
 - [scheduler.c, 170](#)
- [data](#)
 - [vec_st, 21](#)
- [decrement_fd_ref_count](#)
 - [fs_helpers.c, 65](#)
 - [fs_helpers.h, 82](#)

- delete_from_explicit_queue
 - kern_sys_calls.c, [136](#)
 - kern_sys_calls.h, [146](#)
- delete_from_queue
 - kern_sys_calls.c, [136](#)
 - kern_sys_calls.h, [147](#)
- delete_process_from_all_queues
 - scheduler.c, [164](#)
 - scheduler.h, [175](#)
- delete_process_from_all_queues_except_current
 - scheduler.c, [164](#)
 - scheduler.h, [175](#)
- delete_process_from_particular_queue
 - scheduler.c, [165](#)
 - scheduler.h, [176](#)
- det_priorities_arr
 - scheduler.c, [171](#)
- determine_index_in_queue
 - kern_sys_calls.c, [136](#)
 - kern_sys_calls.h, [147](#)
- dir_entry_t, [5](#)
 - firstBlock, [5](#)
 - mtime, [5](#)
 - name, [6](#)
 - perm, [6](#)
 - reserved, [6](#)
 - size, [6](#)
 - type, [6](#)
- ele_dtor_fn
 - vec_st, [21](#)
- execute_command
 - shell.c, [249](#)
- EXITED_NORMALLY
 - signal.h, [183](#)
- EXPECT_COMMANDS
 - parser.h, [242](#)
- EXPECT_INPUT_FILENAME
 - parser.h, [242](#)
- EXPECT_OUTPUT_FILENAME
 - parser.h, [243](#)
- F_APPEND
 - fat_routines.h, [39](#)
- F_READ
 - fat_routines.h, [39](#)
- F_WRITE
 - fat_routines.h, [39](#)
- fat
 - fs_helpers.c, [70](#)
 - fs_helpers.h, [87](#)
- FAT_EOF
 - fat_routines.h, [39](#)
- FAT_FREE
 - fat_routines.h, [39](#)
- fat_routines.c
 - cat, [24](#)
 - chmod, [28](#)
 - cmpctdir, [29](#)
 - cp, [29](#)
 - current_running_pcb, [37](#)
 - ls, [30](#)
 - mkfs, [31](#)
 - mount, [32](#)
 - mv, [33](#)
 - rm, [34](#)
 - touch, [35](#)
 - unmount, [36](#)
- fat_routines.h
 - cat, [41](#)
 - chmod, [45](#)
 - cmpctdir, [46](#)
 - cp, [47](#)
 - F_APPEND, [39](#)
 - F_READ, [39](#)
 - F_WRITE, [39](#)
 - FAT_EOF, [39](#)
 - FAT_FREE, [39](#)
 - ls, [48](#)
 - mkfs, [49](#)
 - mount, [50](#)
 - mv, [51](#)
 - PERM_EXEC, [39](#)
 - PERM_NONE, [40](#)
 - PERM_READ, [40](#)
 - PERM_READ_EXEC, [40](#)
 - PERM_READ_WRITE, [40](#)
 - PERM_READ_WRITE_EXEC, [40](#)
 - PERM_WRITE, [40](#)
 - rm, [53](#)
 - touch, [54](#)
 - TYPE_DIRECTORY, [41](#)
 - TYPE_REGULAR, [41](#)
 - TYPE_SYMLINK, [41](#)
 - TYPE_UNKNOWN, [41](#)
 - unmount, [55](#)
- fat_size
 - fs_helpers.c, [70](#)
 - fs_helpers.h, [87](#)
- fd_entry_t, [7](#)
 - filename, [7](#)
 - first_block, [7](#)
 - in_use, [7](#)
 - mode, [7](#)
 - position, [8](#)
 - ref_count, [8](#)
 - size, [8](#)
- fd_table
 - fs_helpers.c, [70](#)
 - fs_helpers.h, [88](#)
 - pcb_st, [13](#)
- FILE_DESCRIPTOR_TABLE_SIZE
 - kern_pcb.h, [130](#)
- filename
 - fd_entry_t, [7](#)
- fill_buffer_until_full_or_newline
 - shell.c, [251](#)

- find_file
 - fs_helpers.c, 66
 - fs_helpers.h, 82
- findJobByIdOrCurrent
 - shell_built_ins.c, 263
 - shell_built_ins.h, 277
- FINISHED
 - Job.h, 236
- finished_count
 - job_st, 9
- first_block
 - fd_entry_t, 7
- firstBlock
 - dir_entry_t, 5
- free_job_ptr
 - shell.c, 251
- free_pcb
 - kern_pcb.c, 125
 - kern_pcb.h, 131
- free_scheduler_queues
 - scheduler.c, 165
 - scheduler.h, 176
- fs_fd
 - fs_helpers.c, 70
 - fs_helpers.h, 88
- fs_helpers.c
 - add_file_entry, 57
 - allocate_block, 59
 - block_size, 70
 - compact_directory, 59
 - copy_host_to_pennfat, 62
 - copy_pennfat_to_host, 63
 - copy_source_to_dest, 64
 - decrement_fd_ref_count, 65
 - fat, 70
 - fat_size, 70
 - fd_table, 70
 - find_file, 66
 - fs_fd, 70
 - get_free_fd, 67
 - has_executable_permission, 67
 - increment_fd_ref_count, 68
 - init_fd_table, 68
 - is_mounted, 70
 - mark_entry_as_deleted, 69
 - MAX_FDS, 71
 - num_fat_blocks, 71
- fs_helpers.h
 - add_file_entry, 72
 - allocate_block, 74
 - block_size, 87
 - compact_directory, 75
 - copy_host_to_pennfat, 78
 - copy_pennfat_to_host, 79
 - copy_source_to_dest, 80
 - decrement_fd_ref_count, 82
 - fat, 87
 - fat_size, 87
 - fd_table, 88
 - find_file, 82
 - fs_fd, 88
 - get_free_fd, 84
 - has_executable_permission, 84
 - increment_fd_ref_count, 85
 - init_fd_table, 85
 - is_mounted, 88
 - mark_entry_as_deleted, 86
 - MAX_FDS, 88
 - num_fat_blocks, 88
- fs_kfuncs.c
 - current_fg_pid, 100
 - current_running_pcb, 100
 - k_close, 90
 - k_ls, 90
 - k_lseek, 92
 - k_open, 93
 - k_read, 95
 - k_unlink, 96
 - k_write, 97
- fs_kfuncs.h
 - k_close, 102
 - k_ls, 103
 - k_lseek, 105
 - k_open, 106
 - k_read, 109
 - k_unlink, 110
 - k_write, 111
 - SEEK_CUR, 102
 - SEEK_END, 102
 - SEEK_SET, 102
- fs_syscalls.c
 - s_close, 116
 - s_ls, 116
 - s_lseek, 116
 - s_open, 117
 - s_read, 117
 - s_unlink, 117
 - s_write, 118
- fs_syscalls.h
 - s_close, 120
 - s_ls, 120
 - s_lseek, 121
 - s_open, 121
 - s_read, 122
 - s_unlink, 123
 - s_write, 123
 - STDERR_FILENO, 119
 - STDIN_FILENO, 119
 - STDOUT_FILENO, 119
- generate_next_priority
 - scheduler.c, 165
 - scheduler.h, 176
- get_associated_ufunc
 - shell_built_ins.c, 274
- get_free_fd
 - fs_helpers.c, 67

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

- k_proc_cleanup, 132
 - k_proc_create, 133
 - pcb_t, 130
 - remove_child_in_parent, 134
- kern_sys_calls.c
 - current_fg_pid, 143
 - current_pcb, 143
 - current_running_pcb, 143
 - delete_from_explicit_queue, 136
 - delete_from_queue, 136
 - determine_index_in_queue, 136
 - init_func, 137
 - move_pcb_correct_queue, 137
 - one_priority_queue, 144
 - s_cleanup_init_process, 138
 - s_echo, 138
 - s_exit, 139
 - s_kill, 139
 - s_nice, 139
 - s_ps, 140
 - s_sleep, 140
 - s_spawn, 141
 - s_spawn_init, 141
 - s_waitpid, 142
 - sleep_blocked_queue, 144
 - tick_counter, 144
 - two_priority_queue, 144
 - zero_priority_queue, 144
 - zombie_queue, 144
- kern_sys_calls.h
 - delete_from_explicit_queue, 146
 - delete_from_queue, 147
 - determine_index_in_queue, 147
 - init_func, 148
 - move_pcb_correct_queue, 148
 - s_cleanup_init_process, 149
 - s_echo, 149
 - s_exit, 150
 - s_kill, 150
 - s_nice, 151
 - s_ps, 151
 - s_sleep, 152
 - s_spawn, 153
 - s_spawn_init, 154
 - s_waitpid, 154
- length
 - vec_st, 21
- log_fd
 - logger.h, 161
 - pennos.c, 230
 - scheduler.c, 171
- log_generic_event
 - logger.c, 156
 - logger.h, 158
- log_nice_event
 - logger.c, 157
 - logger.h, 160
- log_scheduling_event
 - logger.c, 157
 - logger.h, 160
- logger.c
 - log_generic_event, 156
 - log_nice_event, 157
 - log_scheduling_event, 157
- logger.h
 - log_fd, 161
 - log_generic_event, 158
 - log_nice_event, 160
 - log_scheduling_event, 160
 - tick_counter, 161
- ls
 - fat_routines.c, 30
 - fat_routines.h, 48
- main
 - pennfat.c, 227
 - pennos.c, 230
- mark_entry_as_deleted
 - fs_helpers.c, 69
 - fs_helpers.h, 86
- MAX_BUFFER_SIZE
 - shell.c, 248
- MAX_FDS
 - fs_helpers.c, 71
 - fs_helpers.h, 88
- MAX_LINE_BUFFER_SIZE
 - shell.c, 249
- meta
 - spthread_st, 20
- meta_mutex
 - spthread_meta_st, 17
- MILISEC_IN_NANO
 - spthread.c, 195
- mkfs
 - fat_routines.c, 31
 - fat_routines.h, 49
- mode
 - fd_entry_t, 7
- mount
 - fat_routines.c, 32
 - fat_routines.h, 50
- move_pcb_correct_queue
 - kern_sys_calls.c, 137
 - kern_sys_calls.h, 148
- mtime
 - dir_entry_t, 5
- mv
 - fat_routines.c, 33
 - fat_routines.h, 51
- name
 - dir_entry_t, 6
- next_job_id
 - shell.c, 258
- next_pid
 - kern_pcb.c, 128
- nohang

- stress.c, 186
- stress.h, 188
- num_commands
 - parsed_command, 11
- num_fat_blocks
 - fs_helpers.c, 71
 - fs_helpers.h, 88
- num_pids
 - job_st, 9
- one_priority_queue
 - kern_sys_calls.c, 144
 - scheduler.c, 171
- orphan_child
 - shell_built_ins.c, 264
- output_fd
 - pcb_st, 14
- output_fd_script
 - shell.c, 258
- P_EBADF
 - pennos-errno.h, 190
- P_EBUSY
 - pennos-errno.h, 190
- P_ECLOSE
 - pennos-errno.h, 190
- P_ECOMMAND
 - pennos-errno.h, 190
- P_EEXIST
 - pennos-errno.h, 190
- P_EFS_NOT_MOUNTED
 - pennos-errno.h, 190
- P_EFULL
 - pennos-errno.h, 191
- P_EFUNC
 - pennos-errno.h, 191
- P_EINTR
 - pennos-errno.h, 191
- P_EINVAL
 - pennos-errno.h, 191
- P_ELSEEK
 - pennos-errno.h, 191
- P_EMALLOC
 - pennos-errno.h, 191
- P_EMAP
 - pennos-errno.h, 192
- P_ENOENT
 - pennos-errno.h, 192
- P_ENULL
 - pennos-errno.h, 192
- P_EOPEN
 - pennos-errno.h, 192
- P_EPARSE
 - pennos-errno.h, 192
- P_EPERM
 - pennos-errno.h, 192
- P_EREAD
 - pennos-errno.h, 193
- P_EREDIR
 - pennos-errno.h, 193
- P_ERRNO
 - pennos-errno.c, 189
 - pennos-errno.h, 194
- P_ESIGNAL
 - pennos-errno.h, 193
- P_EUNKNOWN
 - pennos-errno.h, 193
- P_EWRITE
 - pennos-errno.h, 193
- P_INITFAIL
 - pennos-errno.h, 193
- P_NEEDF
 - pennos-errno.h, 194
- P_SIGCONT
 - signal.h, 183
- P_SIGSTOP
 - signal.h, 184
- P_SIGTERM
 - signal.h, 184
- P_WIFEXITED
 - signal.h, 184
- P_WIFSIGNALED
 - signal.h, 184
- P_WIFSTOPPED
 - signal.h, 184
- par_pid
 - pcb_st, 14
- parse_command
 - parser.c, 237
 - parser.h, 244
- parsed_command, 10
 - commands, 11
 - is_background, 11
 - is_file_append, 11
 - num_commands, 11
 - stdin_file, 11
 - stdout_file, 11
- parser.c
 - JUMP_OUT, 237
 - parse_command, 237
 - print_parsed_command, 240
 - print_parser_errcode, 240
- parser.h
 - EXPECT_COMMANDS, 242
 - EXPECT_INPUT_FILENAME, 242
 - EXPECT_OUTPUT_FILENAME, 243
 - parse_command, 244
 - print_parsed_command, 246
 - print_parser_errcode, 247
 - UNEXPECTED_AMPERSAND, 243
 - UNEXPECTED_FILE_INPUT, 243
 - UNEXPECTED_FILE_OUTPUT, 243
 - UNEXPECTED_PIPELINE, 243
- pcb_st, 12
 - child_pcbs, 13
 - cmd_str, 13
 - fd_table, 13

- input_fd, 13
- is_sleeping, 13
- output_fd, 14
- par_pid, 14
- pid, 14
- priority, 14
- process_state, 14
- process_status, 14
- signals, 15
- thread_handle, 15
- time_to_wake, 15
- pcb_t
 - kern_pcb.h, 130
- pennfat.c
 - main, 227
 - PROMPT, 227
- pennos-errno.c
 - P_ERRNO, 189
- pennos-errno.h
 - P_EBADF, 190
 - P_EBUSY, 190
 - P_ECLOSE, 190
 - P_ECOMMAND, 190
 - P_EEXIST, 190
 - P_EFS_NOT_MOUNTED, 190
 - P_EFULL, 191
 - P_EFUNC, 191
 - P_EINTR, 191
 - P_EINVAL, 191
 - P_ELSEEK, 191
 - P_EMALLOC, 191
 - P_EMAP, 192
 - P_ENOENT, 192
 - P_ENULL, 192
 - P_EOPEN, 192
 - P_EPARSE, 192
 - P_EPERM, 192
 - P_EREAD, 193
 - P_EREDIR, 193
 - P_ERRNO, 194
 - P_ESIGNAL, 193
 - P_EUNKNOWN, 193
 - P_EWRITE, 193
 - P_INITFAIL, 193
 - P_NEEDF, 194
- pennos.c
 - log_fd, 230
 - main, 230
 - tick_counter, 230
- perm
 - dir_entry_t, 6
- PERM_EXEC
 - fat_routines.h, 39
- PERM_NONE
 - fat_routines.h, 40
- PERM_READ
 - fat_routines.h, 40
- PERM_READ_EXEC
 - fat_routines.h, 40
- fat_routines.h, 40
- PERM_READ_WRITE
 - fat_routines.h, 40
- PERM_READ_WRITE_EXEC
 - fat_routines.h, 40
- PERM_WRITE
 - fat_routines.h, 40
- pgid
 - job_st, 9
- pid
 - pcb_st, 14
- pids
 - job_st, 10
- position
 - fd_entry_t, 8
- print_parsed_command
 - parser.c, 240
 - parser.h, 246
- print_parser_errcode
 - parser.c, 240
 - parser.h, 247
- priority
 - pcb_st, 14
- process_state
 - pcb_st, 14
- process_status
 - pcb_st, 14
- PROMPT
 - pennfat.c, 227
 - shell.c, 249
- pthread_fn
 - sphthead.c, 197
- ptr_dtor_fn
 - Vec.h, 219
- ptr_t
 - Vec.h, 219
- put_pcb_into_correct_queue
 - scheduler.c, 168
 - scheduler.h, 180
- recur
 - stress.c, 186
 - stress.h, 188
- ref_count
 - fd_entry_t, 8
- remove_child_in_parent
 - kern_pcb.c, 127
 - kern_pcb.h, 134
- reserved
 - dir_entry_t, 6
- rm
 - fat_routines.c, 34
 - fat_routines.h, 53
- RUNNING
 - Job.h, 236
- s_cleanup_init_process
 - kern_sys_calls.c, 138
 - kern_sys_calls.h, 149

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

- shell.c, 252
- shell.h, 259
- shell.c
 - current_fg_pid, 257
 - execute_command, 249
 - fill_buffer_until_full_or_newline, 251
 - free_job_ptr, 251
 - input_fd_script, 257
 - is_append, 257
 - job_list, 258
 - MAX_BUFFER_SIZE, 248
 - MAX_LINE_BUFFER_SIZE, 249
 - next_job_id, 258
 - output_fd_script, 258
 - PROMPT, 249
 - script_fd, 258
 - setup_terminal_signal_handlers, 251
 - shell, 252
 - shell_sigint_handler, 255
 - shell_sigstp_handler, 255
 - u_execute_command, 255
 - u_read_and_execute_script, 256
- shell.h
 - is_append, 261
 - shell, 259
- shell_built_ins.c
 - current_fg_pid, 274
 - findJobByIdOrCurrent, 263
 - get_associated_ufunc, 274
 - job_list, 275
 - orphan_child, 264
 - u_bg, 264
 - u_busy, 265
 - u_cat, 265
 - u_chmod, 266
 - u_cp, 266
 - u_echo, 266
 - u_fg, 267
 - u_jobs, 268
 - u_kill, 269
 - u_logout, 269
 - u_ls, 270
 - u_man, 270
 - u_mv, 271
 - u_nice, 271
 - u_nice_pid, 271
 - u_orphanify, 272
 - u_ps, 272
 - u_rm, 272
 - u_sleep, 273
 - u_touch, 273
 - u_zombify, 273
 - zombie_child, 274
- shell_built_ins.h
 - findJobByIdOrCurrent, 277
 - u_bg, 278
 - u_busy, 279
 - u_cat, 279
 - u_chmod, 279
 - u_cp, 280
 - u_echo, 280
 - u_fg, 281
 - u_jobs, 282
 - u_kill, 283
 - u_logout, 283
 - u_ls, 284
 - u_man, 284
 - u_mv, 285
 - u_nice, 285
 - u_nice_pid, 286
 - u_orphanify, 286
 - u_ps, 287
 - u_rm, 287
 - u_sleep, 287
 - u_touch, 288
 - u_zombify, 288
- shell_sigint_handler
 - shell.c, 255
- shell_sigstp_handler
 - shell.c, 255
- shutup_mutex
 - sphthead_signal_args_st, 18
- signal
 - sphthead_signal_args_st, 19
- signal.h
 - CONT_BY_SIG, 183
 - EXITED_NORMALLY, 183
 - P_SIGCONT, 183
 - P_SIGSTOP, 184
 - P_SIGTERM, 184
 - P_WIFEXITED, 184
 - P_WIFSIGNALED, 184
 - P_WIFSTOPPED, 184
 - STOPPED_BY_SIG, 185
 - TERM_BY_SIG, 185
- signals
 - pcb_st, 15
- SIGPTHD
 - sphthead.h, 203
- size
 - dir_entry_t, 6
 - fd_entry_t, 8
- sleep_blocked_queue
 - kern_sys_calls.c, 144
 - scheduler.c, 171
- sphthead.c
 - _GNU_SOURCE, 195
 - MILISEC_IN_NANO, 195
 - pthread_fn, 197
 - sphthead_cancel, 197
 - sphthead_continue, 197
 - sphthead_create, 198
 - sphthead_disable_interrupts_self, 199
 - sphthead_enable_interrupts_self, 199
 - sphthead_equal, 200
 - sphthead_exit, 200

- spthread_fwd_args, 197
- spthread_join, 200
- spthread_meta_t, 197
- SPTHREAD_RUNNING_STATE, 196
- spthread_self, 200
- SPTHREAD_SIG_CONTINUE, 196
- SPTHREAD_SIG_SUSPEND, 196
- spthread_signal_args, 197
- spthread_suspend, 201
- spthread_suspend_self, 201
- SPTHREAD_SUSPENDED_STATE, 196
- SPTHREAD_TERMINATED_STATE, 196
- spthread.h
 - SIGPTH, 203
 - spthread_cancel, 204
 - spthread_continue, 204
 - spthread_create, 205
 - spthread_disable_interrupts_self, 205
 - spthread_enable_interrupts_self, 206
 - spthread_equal, 206
 - spthread_exit, 206
 - spthread_join, 207
 - spthread_meta_t, 203
 - spthread_self, 207
 - spthread_suspend, 207
 - spthread_suspend_self, 208
 - spthread_t, 203
- spthread_cancel
 - spthread.c, 197
 - spthread.h, 204
- spthread_continue
 - spthread.c, 197
 - spthread.h, 204
- spthread_create
 - spthread.c, 198
 - spthread.h, 205
- spthread_disable_interrupts_self
 - spthread.c, 199
 - spthread.h, 205
- spthread_enable_interrupts_self
 - spthread.c, 199
 - spthread.h, 206
- spthread_equal
 - spthread.c, 200
 - spthread.h, 206
- spthread_exit
 - spthread.c, 200
 - spthread.h, 206
- spthread_fwd_args
 - spthread.c, 197
- spthread_fwd_args_st, 15
 - actual_arg, 16
 - actual_routine, 16
 - child_meta, 16
 - setup_cond, 16
 - setup_done, 16
 - setup_mutex, 17
- spthread_join
 - spthread.c, 200
 - spthread.h, 207
- spthread_meta_st, 17
 - meta_mutex, 17
 - state, 17
 - suspend_set, 18
- spthread_meta_t
 - spthread.c, 197
 - spthread.h, 203
- SPTHREAD_RUNNING_STATE
 - spthread.c, 196
- spthread_self
 - spthread.c, 200
 - spthread.h, 207
- SPTHREAD_SIG_CONTINUE
 - spthread.c, 196
- SPTHREAD_SIG_SUSPEND
 - spthread.c, 196
- spthread_signal_args
 - spthread.c, 197
- spthread_signal_args_st, 18
 - ack, 18
 - shutup_mutex, 18
 - signal, 19
- spthread_st, 19
 - meta, 20
 - thread, 20
- spthread_suspend
 - spthread.c, 201
 - spthread.h, 207
- spthread_suspend_self
 - spthread.c, 201
 - spthread.h, 208
- SPTHREAD_SUSPENDED_STATE
 - spthread.c, 196
- spthread_t
 - spthread.h, 203
- SPTHREAD_TERMINATED_STATE
 - spthread.c, 196
- SRC/fs/fat_routines.c, 23
- SRC/fs/fat_routines.h, 37
- SRC/fs/fs_helpers.c, 56
- SRC/fs/fs_helpers.h, 71
- SRC/fs/fs_kfuncs.c, 89
- SRC/fs/fs_kfuncs.h, 101
- SRC/fs/fs_syscalls.c, 115
- SRC/fs/fs_syscalls.h, 118
- SRC/kernel/kern_pcb.c, 124
- SRC/kernel/kern_pcb.h, 129
- SRC/kernel/kern_sys_calls.c, 134
- SRC/kernel/kern_sys_calls.h, 145
- SRC/kernel/logger.c, 156
- SRC/kernel/logger.h, 158
- SRC/kernel/scheduler.c, 162
- SRC/kernel/scheduler.h, 172
- SRC/kernel/signal.c, 182
- SRC/kernel/signal.h, 183
- SRC/kernel/stress.c, 185

- SRC/kernel/stress.h, 187
- SRC/lib/pennos-errno.c, 188
- SRC/lib/pennos-errno.h, 189
- SRC/lib/spthread.c, 194
- SRC/lib/spthread.h, 202
- SRC/lib/Vec.c, 209
- SRC/lib/Vec.h, 217
- SRC/pennfat.c, 227
- SRC/pennos.c, 229
- SRC/shell/builtins.c, 231
- SRC/shell/builtins.h, 233
- SRC/shell/Job.h, 234
- SRC/shell/parser.c, 236
- SRC/shell/parser.h, 241
- SRC/shell/shell.c, 247
- SRC/shell/shell.h, 258
- SRC/shell/shell_built_ins.c, 262
- SRC/shell/shell_built_ins.h, 275
- state
 - job_st, 10
 - spthread_meta_st, 17
- STDERR_FILENO
 - fs_syscalls.h, 119
- stdin_file
 - parsed_command, 11
- STDIN_FILENO
 - fs_syscalls.h, 119
- stdout_file
 - parsed_command, 11
- STDOUT_FILENO
 - fs_syscalls.h, 119
- STOPPED
 - Job.h, 236
- STOPPED_BY_SIG
 - signal.h, 185
- stress.c
 - crash, 186
 - hang, 186
 - nohang, 186
 - recur, 186
- stress.h
 - crash, 187
 - hang, 187
 - nohang, 188
 - recur, 188
- 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
 - pennos.c, 230
 - scheduler.c, 171
- time_to_wake
 - pcb_st, 15
- touch
 - fat_routines.c, 35
 - fat_routines.h, 54
- two_priority_queue
 - kern_sys_calls.c, 144
 - scheduler.c, 171
- type
 - dir_entry_t, 6
- TYPE_DIRECTORY
 - fat_routines.h, 41
- TYPE_REGULAR
 - fat_routines.h, 41
- TYPE_SYMLINK
 - fat_routines.h, 41
- TYPE_UNKNOWN
 - fat_routines.h, 41
- u_bg
 - shell_built_ins.c, 264
 - shell_built_ins.h, 278
- u_busy
 - shell_built_ins.c, 265
 - shell_built_ins.h, 279
- u_cat
 - shell_built_ins.c, 265
 - shell_built_ins.h, 279
- u_chmod
 - shell_built_ins.c, 266
 - shell_built_ins.h, 279
- u_cp
 - shell_built_ins.c, 266
 - shell_built_ins.h, 280
- u_echo
 - shell_built_ins.c, 266
 - shell_built_ins.h, 280
- u_execute_command
 - shell.c, 255
- u_fg
 - shell_built_ins.c, 267
 - shell_built_ins.h, 281
- u_jobs
 - shell_built_ins.c, 268
 - shell_built_ins.h, 282
- 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
- u_mv
 - shell_built_ins.c, 271

- shell_built_ins.h, [285](#)
- u_nice
 - shell_built_ins.c, [271](#)
 - shell_built_ins.h, [285](#)
- u_nice_pid
 - shell_built_ins.c, [271](#)
 - shell_built_ins.h, [286](#)
- u_orphanify
 - shell_built_ins.c, [272](#)
 - shell_built_ins.h, [286](#)
- u_perror
 - builtins.c, [231](#)
 - builtins.h, [233](#)
- u_ps
 - shell_built_ins.c, [272](#)
 - shell_built_ins.h, [287](#)
- u_read_and_execute_script
 - shell.c, [256](#)
- u_rm
 - shell_built_ins.c, [272](#)
 - shell_built_ins.h, [287](#)
- u_sleep
 - shell_built_ins.c, [273](#)
 - shell_built_ins.h, [287](#)
- u_touch
 - shell_built_ins.c, [273](#)
 - shell_built_ins.h, [288](#)
- u_zombify
 - shell_built_ins.c, [273](#)
 - shell_built_ins.h, [288](#)
- UNEXPECTED_AMPERSAND
 - parser.h, [243](#)
- UNEXPECTED_FILE_INPUT
 - parser.h, [243](#)
- UNEXPECTED_FILE_OUTPUT
 - parser.h, [243](#)
- UNEXPECTED_PIPELINE
 - parser.h, [243](#)
- unmount
 - fat_routines.c, [36](#)
 - fat_routines.h, [55](#)
- Vec
 - Vec.h, [219](#)
- Vec.c
 - vec_clear, [209](#)
 - vec_destroy, [210](#)
 - vec_erase, [210](#)
 - vec_erase_no_deletor, [211](#)
 - vec_get, [212](#)
 - vec_insert, [212](#)
 - vec_new, [213](#)
 - vec_pop_back, [214](#)
 - vec_push_back, [214](#)
 - vec_resize, [215](#)
 - vec_set, [216](#)
- Vec.h
 - ptr_dtor_fn, [219](#)
 - ptr_t, [219](#)
- Vec, [219](#)
- vec_capacity, [218](#)
- vec_clear, [219](#)
- vec_destroy, [220](#)
- vec_erase, [220](#)
- vec_erase_no_deletor, [221](#)
- vec_get, [222](#)
- vec_insert, [222](#)
- vec_is_empty, [218](#)
- vec_len, [218](#)
- vec_new, [223](#)
- vec_pop_back, [224](#)
- vec_push_back, [224](#)
- vec_resize, [225](#)
- vec_set, [226](#)
- vec_capacity
 - Vec.h, [218](#)
- vec_clear
 - Vec.c, [209](#)
 - Vec.h, [219](#)
- vec_destroy
 - Vec.c, [210](#)
 - Vec.h, [220](#)
- vec_erase
 - Vec.c, [210](#)
 - Vec.h, [220](#)
- vec_erase_no_deletor
 - Vec.c, [211](#)
 - Vec.h, [221](#)
- vec_get
 - Vec.c, [212](#)
 - Vec.h, [222](#)
- vec_insert
 - Vec.c, [212](#)
 - Vec.h, [222](#)
- vec_is_empty
 - Vec.h, [218](#)
- vec_len
 - Vec.h, [218](#)
- vec_new
 - Vec.c, [213](#)
 - Vec.h, [223](#)
- vec_pop_back
 - Vec.c, [214](#)
 - Vec.h, [224](#)
- vec_push_back
 - Vec.c, [214](#)
 - Vec.h, [224](#)
- vec_resize
 - Vec.c, [215](#)
 - Vec.h, [225](#)
- vec_set
 - Vec.c, [216](#)
 - Vec.h, [226](#)
- vec_st, [20](#)
 - capacity, [20](#)
 - data, [21](#)
 - ele_dtor_fn, [21](#)

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](#)