PennOS Companion Document

Table of Contents

---

1. File Structure

```
.
├── Makefile
├── bin/
│   ├── pennfat
│   └── pennos
├── doc/
│   ├── Companion Document.pdf
│   └── README.md
├── log/
├── src/
│   ├── fat_kernel.c
│   ├── fat_kernel.h
│   ├── fat_syscalls.c
│   ├── fat_syscalls.h
│   ├── pennfat.c
│   ├── pennos.c
│   ├── process.c
│   ├── process.h
│   ├── scheduler.c
│   ├── scheduler.h
│   ├── syscall.c
│   ├── syscall.h
│   ├── user_function.c
│   ├── user_function.h
│   └── util/
│       ├── Vec.c
│       ├── Vec.h
│       ├── job.c
│       ├── job.h
│       ├── p_errno.c
│       ├── p_errno.h
│       ├── p_handler.c
│       ├── p_handler.h
│       ├── p_signal.c
│       ├── p_signal.h
│       ├── panic.c
│       ├── panic.h
│       ├── parser.c
```

```
│       ├── parser.h
│       ├── queue.c
│       ├── queue.h
│       ├── spthread.c
│       ├── spthread.h
│       ├── stress.c
│       ├── stress.h
│       ├── struct.c
│       └── struct.h
└── test/
    └── sched-demo.c
```

---

2. Data Structures

PCB (Process Control Block)

The core data structure representing a process.

```c
typedef struct pcb {
  // Process identity
  spthread_t process;            // spthread handle for the process
  char cmd_name[MAX_NAME_LEN];   // process name/command brief
  char** args;                   // deep-copied arguments
  pid_t pid;                     // process id
  pstate_t state;                // process state
  int prio;                      // Priority: 0, 1, or 2
  int wake_tick;                 // used while sleeping (in clock ticks)
  bool stopped_reported;         // if STOPPED state has been reported to waitpid

  pid_t ppid;            // parent's process id
  struct pcb* parent;    // parent's pcb pointer
  Vec childs;            // vector storing childs of the process

  int fd_table[MAX_FD];  // local fd table, stores kfd indexes
  pexit_t exit_status;   // exit status
} pcb_t;
```

File: src/util/struct.h

Process States and Exit Status

Enumerations defining the possible states of a process and its exit status.

```c
/** PennOS process state */
typedef enum {
    P_READY,    // Ready to run, waiting for scheduler
    P_RUNNING,  // Currently executing on a processor
    P_BLOCKED,  // Blocked, waiting for an event (e.g., sleep, I/O)
    P_STOPPED,  // Stopped by a signal
    P_ZOMBIE    // Terminated but not yet reaped by parent
} pstate_t;
```

```c
/** PennOS process exit status */
typedef enum {
  P_EXIT_NONE,      // Process is still running (default)
  P_EXIT_EXITED,    // Process exited normally via exit()
  P_EXIT_SIGNALED,  // Process was terminated by a signal
  P_EXIT_STOPPED    // Process was stopped by a signal
} pexit_t;
```

File: `src/util/struct.h`

Directory Entry

Represents a file's metadata within the PennFAT filesystem. Each directory entry is stored in the filesystem and contains all the information needed to locate and access a file. The root directory and any subdirectories consist of an array of these entries.

```c
typedef struct {
  char name[32];          // Null-terminated filename (includes extension if any)
  uint32_t size;          // File size in bytes (actual data, not including
metadata)
  uint16_t firstBlock;    // Starting block number in the data region
  uint8_t type;           // File type: 1 = Regular file, 2 = Directory file
  uint8_t perm;           // Permission bits (refer to specification for rwx
encoding)
  time_t mtime;           // Last modification time (Unix timestamp)
  char reserved[16];      // Reserved space for future use or alignment (currently
unused)
} dir_entry_t;
```

File: `src/fat_kernel.h`

Open File Entry

Represents an open file instance in the kernel. This structure is stored in the Global File Descriptor Table (GDT) and contains metadata along with per-descriptor state for open files.

```c
typedef struct open_file {
  char name[MAX_NAME_LEN];  // Cached file name
  uint32_t size;            // Cached file size
  uint8_t perm;             // Cached permissions
  uint16_t first_block;     // Fast reference to file's first block in FAT chain
  off_t dirent_offset;      // Fast reference to directory entry location on disk
(for updates)
  uint64_t offset;          // File descriptor-specific read/write position (current
seek offset)
  uint8_t flag;             // File descriptor-specific flags: F_READ, F_WRITE,
F_APPEND
} open_file_t;
```

File: `src/util/struct.h`

Job Entry

Represents a job in the shell's job control system. It maps a user-facing job ID to a process ID and tracks the job's state and command line.

```
typedef struct job {
  int job_id;        // Small integer assigned by the job subsystem
  pid_t pid;         // Process id associated with this job
  pcb_t* pcb;        // Pointer to the process control block
  char cmd[64];      // Short copy of the command line
  job_state_t state; // Current job state (RUNNING, STOPPED, BACKGROUND, DONE)
  bool used;         // Flag indicating this table slot is occupied
} job_t;
```

File: `src/util/job.h`

Job State

Enumeration defining the possible states of a job.

```
typedef enum {
  JOB_RUNNING,    // The job's process is currently running
  JOB_STOPPED,    // The job has been stopped
  JOB_BACKGROUND, // The job is running in the background
  JOB_DONE        // The job has terminated and can be removed
} job_state_t;
```

File: `src/util/job.h`

Spawn Wrapper Arguments

A structure used to pass arguments and redirection configurations to the child process entry function during a `spawn` system call.

```
typedef struct {
  void* (*func)(void*); // Function to be executed by the child
  char** argv;          // Command arguments
  char* stdin_file;     // Input redirection file
  char* stdout_file;    // Output redirection file
  int is_append;        // Append mode flag for output
  int saved_stdin;      // Saved STDIN file descriptor (for restoration)
  int saved_stdout;     // Saved STDOUT file descriptor (for restoration)
} spawn_wrapper_args_t;
```

File: `src/syscall.h`

Parsed Command

Represents the result of parsing a shell command line. Contains flags for background execution, redirection, and the command arguments. This part is identical to what was provided to us.

```
struct parsed_command {
  int is_background;   // Background execution flag (&)
  int is_file_append;  // Append output flag (>>)
  char* stdin_file;    // Input redirection filename (<)
  char* stdout_file;   // Output redirection filename (> or >>)
  size_t num_commands; // Number of commands in pipeline
  char** commands[];   // Flexible array of command arguments
};
```

File: src/util/parser.h

Vector (Vec)

A generic dynamic array implementation used throughout the kernel for managing collections, such as priority queues and child process lists.

```
typedef struct vec_st {
  ptr_t* data;             // Array of pointers to elements
  size_t length;           // Current number of elements
  size_t capacity;         // Current allocated capacity
  ptr_dtor_fn ele_dtor_fn; // Function to clean up elements on removal
} Vec;
```

File: src/util/Vec.h

PennOS Error Codes

PennOS uses a global `P_ERRNO` variable to report errors, similar to the standard C `errno`. System calls set this variable when they encounter an error, allowing user programs to check what went wrong. The `u_perror()` function can be used to print human-readable error messages.

```
typedef enum {
  P_NO_ERR = 0,  // No error

  /* Generic errors */
  P_EPERM,   // Operation not permitted
  P_EINVAL,  // Invalid argument (e.g., invalid mode flags, bad parameters)
  P_ENOMEM,  // Out of memory (malloc/allocation failure)

  /* Process-related errors */
  P_EPID,     // Process does not exist (invalid PID)
  P_ECHILD,   // No child process available to wait on
  P_ESRCH,    // No such process (kill failed to find target)
  P_ETHREAD,  // Thread creation failed (spthread error)

  /* File system-related errors */
  P_ENOENT,   // No such file or directory
  P_EEXIST,   // File already exists (cannot create duplicate)
  P_EISDIR,   // Is a directory (when operation expects a file)
  P_EBADF,    // Invalid file descriptor (bad FD or not open)
  P_EIO,      // I/O error (disk read/write failure)
```

```
  P_ENOSPC,   // No space left on device (disk full)
  P_EROFS,    // Read-only file system (write attempted on read-only FS)
  P_ENODEV,   // No such device (filesystem not mounted)
  P_ENFILE,   // File table overflow (global FD table full)
  P_EBUSY,    // Device or resource busy (file in use, cannot delete)
  P_EACCES,   // Permission denied (insufficient permissions)
  P_EMFILE,   // Too many open files (process FD limit reached)

  /* Signal errors */
  P_SIGINT,   // Failed to set SIGINT handler
  P_SIGTSTP,  // Failed to set SIGTSTP handler

  /* Other errors */
  P_ENAMETOOLONG, // File name too long (exceeds MAX_NAME_LEN)
  P_E2BIG,        // Argument list too long

  P_ERR_MAX  // Sentinel: number of error codes (not a real error)
} p_errno_t;
```

Usage:

- System calls return `-1` or `NULL` on error and set `P_ERRNO` to indicate the specific error
- User programs should check the return value and then inspect `P_ERRNO` to determine the cause
- Use `u_perror(const char* msg)` to print formatted error messages to stderr

File: `src/util/p_errno.h`

Signals

PennOS implements a simplified signal mechanism for inter-process communication and process control. Signals are asynchronous notifications sent to processes to indicate events or request state changes.

```
typedef enum {
  P_SIGTERM,  // Terminate signal - requests process termination
  P_SIGSTOP,  // Stop signal - suspends process execution
  P_SIGCONT,  // Continue signal - resumes a stopped process
  P_SIGCHLD   // Child signal - notifies parent of child state change
} psignal_t;
```

Signal Descriptions:

- **P_SIGTERM**: Requests graceful termination of a process. When delivered via `s_kill()`, the target process's state is changed to `P_ZOMBIE` and its exit status is set to `P_EXIT_SIGNALED`. The process will remain as a zombie until reaped by its parent via `s_waitpid()`.
- **P_SIGSTOP**: Suspends process execution. The target process's state is changed to `P_STOPPED`, removing it from the scheduler's ready queues. The process will not be scheduled for execution until it receives `P_SIGCONT`.
- **P_SIGCONT**: Resumes a stopped process. If the target process is in the `P_STOPPED` state, it is moved back to the ready queue at its original priority level and becomes eligible for scheduling.

- **P_SIGCHLD**: Notifies a parent process that one of its children has changed state (terminated or stopped). This is used internally by the kernel; user programs typically don't send this signal directly but detect child state changes via `s_waitpid()`.

Signal Delivery:

- Signals are delivered via the `s_kill(pid_t pid, int signal)` system call
- The kernel function `k_signal_deliver(pcb_t* proc, psignal_t signal)` handles the actual signal processing
- Signals take effect immediately on the target process (no signal queuing or masking in PennOS)
- Invalid signals or attempts to signal non-existent processes result in `P_EINVAL` or `P_ESRCH` errors

Wait Status Macros: After a child process terminates or stops, `s_waitpid()` returns status information that can be inspected using these macros:

- `P_WIFEXITED(status)`: Returns true if child exited normally
- `P_WIFSIGNALED(status)`: Returns true if child was terminated by a signal
- `P_WIFSTOPPED(status)`: Returns true if child was stopped

File: `src/util/p_signal.h`, `src/util/p_signal.c`

---

3. System API

This section provides a detailed breakdown of the system calls that form the core of PennOS, organized by their source file.

`src/fat_syscalls.c`

This file implements the system call interface for filesystem operations.

- `int s_open(const char* fname, int mode)`
  - **Description**: Opens or creates a file in the PennFAT filesystem. If the file exists, it's opened with the specified mode. If it doesn't exist and write/append mode is specified, a new file is created.
  - **Parameters**:
    - `fname`: The name of the file.
    - `mode`: A bitmask specifying access mode (`F_READ`, `F_WRITE`, `F_APPEND`).
  - **Return Value**: Returns a file descriptor on success, or `-1` on error. `P_ERRNO` is set accordingly.
  - **Error Conditions**:
    - `P_ENODEV` if no filesystem is mounted
    - `P_EINVAL` if mode is invalid
    - `P_ENFILE` if the global file descriptor table is full
    - `P_ENOSPC` if the disk is full (when creating a file in a full directory)
    - `P_EBUSY` if trying to open a file for write/append that is already open for write/append
    - `P_ENOENT` if the file doesn't exist in read mode
    - `P_EISDIR` if trying to open a directory as a file
    - `P_EACCES` if insufficient permissions
    - `P_EMFILE` if process file descriptor table is full

        - `P_EPID` if current process cannot be found
        - `P_ENOMEM` if memory allocation fails

- `ssize_t s_read(int fd, int n, char* buf)`
    - **Description**: Reads data from an open file. Reads up to `n` bytes from the file, starting at the file's current offset. The file offset is advanced by the number of bytes read. Can also read from STDIN (fd 0).
    - **Parameters**:
        - `fd` : The file descriptor.
        - `n` : The maximum number of bytes to read.
        - `buf` : The buffer to store the read data.
    - **Return Value**: Returns the number of bytes read on success, `0` on end-of-file, or `-1` on error. `P_ERRNO` is set.
    - **Error Conditions**:
        - `P_EBADF` if `fd` is invalid
        - `P_EACCES` if the file was not opened with read permission (`F_READ` flag)
        - `P_EIO` if disk read operation fails

- `ssize_t s_write(int fd, const char* str, int n)`
    - **Description**: Writes data to an open file. Writes `n` bytes from `str` to the file, starting at the file's current offset (or at the end if opened with `F_APPEND`). File will grow by allocating new blocks as needed. Can also write to STDOUT (fd 1) or STDERR (fd 2).
    - **Parameters**:
        - `fd` : The file descriptor.
        - `str` : The buffer containing data to write.
        - `n` : The number of bytes to write.
    - **Return Value**: Returns the number of bytes written on success, or `-1` on error. `P_ERRNO` is set.
    - **Error Conditions**:
        - `P_EBADF` if `fd` is invalid or not in the process's file descriptor table
        - `P_EACCES` if the file was not opened with write permission (`F_WRITE` or `F_APPEND` flag)
        - `P_ENOSPC` if the disk is full (no free blocks available for allocation)
        - `P_EIO` if disk write operation fails

- `int s_close(int fd)`
    - **Description**: Closes an open file descriptor. Removes the fd from the process's file descriptor table. When the reference count of the file reaches zero, updates the directory entry on disk with current metadata (size, mtime) and releases the global file descriptor table entry. If the file was previously unlinked but still open, it will be fully deleted when the last fd is closed.
    - **Parameters**:
        - `fd` : The file descriptor to close.
    - **Return Value**: Returns `0` on success, or `-1` on error. `P_ERRNO` is set.
    - **Error Conditions**:
        - `P_EBADF` if `fd` is invalid or not in the process's file descriptor table
        - `P_ENODEV` if no filesystem is mounted

        - `P_EIO` if updating the directory entry on disk fails

- `off_t s_lseek(int fd, int offset, int whence)`
    - **Description**: Changes the read/write offset of an open file. If the file is opened in write mode and seeking beyond the current file size, the file's metadata size will be updated (though actual blocks are only allocated during write operations).
    - **Parameters**:
        - `fd` : The file descriptor.
        - `offset` : The offset value (can be negative for `F_SEEK_CUR` and `F_SEEK_END`).
        - `whence` : The directive for setting the offset:
            - `F_SEEK_SET` (0): Set offset to `offset` bytes from beginning
            - `F_SEEK_CUR` (1): Set offset to current position plus `offset`
            - `F_SEEK_END` (2): Set offset to file size plus `offset`
    - **Return Value**: Returns `0` (FS_SUCCESS) on success, or `-1` on error. `P_ERRNO` is set.
    - **Error Conditions**:
        - `P_ENODEV` if no filesystem is mounted
        - `P_EBADF` if `fd` is invalid or not in the global file descriptor table
        - `P_EINVAL` if `whence` is not a valid value, or if the resulting offset would be negative

- `int s_unlink(const char* fname)`
    - **Description**: Deletes a file from the filesystem. If the file is not currently open, immediately frees the FAT chain and marks the directory entry as deleted (name[0] = '\0'). If the file is still open, marks it for deferred deletion (name[0] = 2); the file will be fully deleted when the last file descriptor closes.
    - **Parameters**:
        - `fname` : The name of the file to delete.
    - **Return Value**: Returns `0` on success, or `-1` on error. `P_ERRNO` is set.
    - **Error Conditions**:
        - `P_ENODEV` if no filesystem is mounted
        - `P_ENOENT` if the file does not exist
        - `P_EISDIR` if trying to unlink a directory (type 2)
        - `P_EIO` if reading/writing directory entry fails

- `int s_ls(const char* filename)`
    - **Description**: Lists the contents of the root directory to standard output, showing file permissions, size, modification time, and name. Uses `s_write()` for output to support redirection. If a filename is provided, displays information for that specific file; otherwise lists all files in the root directory.
    - **Parameters**:
        - `filename` : File to display or NULL to list all files in root directory.
    - **Return Value**: Returns `0` on success, or `-1` on error. `P_ERRNO` is set.
    - **Error Conditions**:
        - `P_ENODEV` if no filesystem is mounted
        - `P_ENOENT` if the specified filename doesn't exist
        - `P_EIO` if disk read operation fails

- `int s_cat(char** args)`
  - **Description**: Concatenates and displays file contents. With no arguments, reads from STDIN and writes to STDOUT. With file arguments, opens each file and writes its contents to STDOUT. Uses `s_read()` and `s_write()` to support proper file descriptor handling and redirection.
  - **Parameters**:
    - `args` : Argument array where `args[0]` is "cat" and `args[1]` onwards are filenames (or NULL for STDIN).
  - **Return Value**: Returns `0` on success, or `-1` on error. `P_ERRNO` is set.
  - **Error Conditions**:
    - `P_ENOENT` if a file doesn't exist
    - `P_EBADF` if file descriptor is invalid
    - `P_EACCES` if file lacks read permission
    - `P_EIO` if read or write operation fails
- `int s_mv(const char* src, const char* dest)`
  - **Description**: Renames a file from `src` to `dest` . If the destination file already exists, it will be unlinked first (subject to write permissions and whether it's currently open). Updates the modification time of the renamed file.
  - **Parameters**:
    - `src` : The current name of the file.
    - `dest` : The new name for the file.
  - **Return Value**: Returns `0` on success, or `-1` on error. `P_ERRNO` is set.
  - **Error Conditions**:
    - `P_ENODEV` if no filesystem is mounted
    - `P_ENOENT` if the source file doesn't exist
    - `P_EACCES` if source file lacks read permission (bit 2) or destination file lacks write permission (bit 1) when overwriting
    - `P_EIO` if disk read/write operation fails
- `int s_cp(char** args)`
  - **Description**: Copies files between PennFAT filesystem and host filesystem, or within PennFAT. Supports three modes: (1) PennFAT to PennFAT: `cp SOURCE DEST` , (2) Host to PennFAT: `cp -h SOURCE DEST` , (3) PennFAT to Host: `cp SOURCE -h DEST` . Creates or truncates the destination file.
  - **Parameters**:
    - `args` : Argument array with format depending on mode. `args[0]` is "cp".
  - **Return Value**: Returns `0` on success, or `-1` on error. `P_ERRNO` is set.
  - **Error Conditions**:
    - `P_ENODEV` if no filesystem is mounted
    - `P_EINVAL` if arguments are missing or invalid
    - `P_ENOENT` if source file doesn't exist
    - `P_ENOSPC` if disk is full when creating destination
    - `P_EIO` if read or write operation fails
- `int s_chmod(const char* fname, int mode)`

    - **Description**: Changes the permission bits of a file. Supports three operation modes encoded in the high bits of `mode`: add permissions (0x80), remove permissions (0x40), or assign permissions (0x20 or numeric). The lower 3 bits specify read (4), write (2), and execute (1) permissions. Updates the file's modification time.
    - **Parameters**:
        - `fname`: The name of the file to modify.
        - `mode`: Permission specification with operation flags and permission bits (0-7 for rwx).
    - **Return Value**: Returns `0` on success, or `-1` on error. `P_ERRNO` is set.
    - **Error Conditions**:
        - `P_ENODEV` if no filesystem is mounted
        - `P_ENOENT` if the file doesn't exist
        - `P_EIO` if disk read/write operation fails

- `int s_check_executable(const char* fname)`
    - **Description**: Checks if a file has execute permissions. This is typically used by the shell or spawn to verify if a program can be run.
    - **Parameters**:
        - `fname`: The name of the file to check.
    - **Return Value**: Returns `0` (FS_SUCCESS) if the file is executable, or `-1` on error. `P_ERRNO` is set.
    - **Error Conditions**:
        - `P_EACCES` if the file does not have execute permission
        - `P_ENOENT` if the file does not exist
        - `P_ENODEV` if no filesystem is mounted

`src/syscall.c`

This file implements the system call interface for process management.

- `pid_t s_spawn(void* (*func)(void*), char* argv[], const char* stdin_file, const char* stdout_file, int is_append)`
    - **Description**: Creates a new child process that executes the specified function. Allocates a new PCB, inherits file descriptors from the parent, and optionally redirects stdin/stdout to files. The child process is assigned priority 1 by default and added to the ready queue. Supports file redirection through a wrapper function that handles opening/closing redirection files. Prevents input/output conflict when the same file is used for both stdin and stdout in append mode.
    - **Parameters**:
        - `func`: A pointer to the function the new process will execute.
        - `argv`: An array of string arguments for the new process (argv[0] is used as the process name).
        - `stdin_file`: Filename to redirect stdin from (NULL for no redirection).
        - `stdout_file`: Filename to redirect stdout to (NULL for no redirection).
        - `is_append`: If non-zero, stdout redirection opens file in append mode; otherwise truncates.
    - **Return Value**: Returns the PID of the new child process on success, or `-1` on error. `P_ERRNO` is set.

    - **Error Conditions**:
        - `P_ENOMEM` if PCB allocation or wrapper argument allocation fails
        - `P_ETHREAD` if spthread creation fails

- `pid_t s_waitpid(pid_t pid, int* wstatus, bool nohang)`
    - **Description**: Waits for a child process to change state (terminate or stop). Can wait for a specific child (if `pid` > 0) or any child (if `pid` == -1). Operates in blocking mode by default, suspending the parent until a child state change occurs. In non-blocking mode (`nohang` = true), returns immediately if no child has changed state. When a child terminates, reaps the zombie and returns status information. For stopped children, returns status only once (tracked by `stopped_reported` flag).
    - **Parameters**:
        - `pid` : The PID of the child to wait for, or -1 to wait for any child.
        - `wstatus` : A pointer to an integer where the child's exit status will be stored (W_EXITED, W_SIGNALED, or W_STOPPED).
        - `nohang` : If `true` , the call returns immediately if no child has changed state. If `false` , it blocks until a child changes state.
    - **Return Value**: Returns the PID of the child whose state changed, `0` if `nohang` is true and no child is ready, or `-1` on error. `P_ERRNO` is set.
    - **Error Conditions**:
        - `P_ECHILD` if the calling process has no children to wait for
        - `P_EINVAL` if current process cannot be found

- `int s_kill(pid_t pid, int signal)`
    - **Description**: Sends a signal to a process, causing an immediate state change. Signal 0 (P_SIGTERM) terminates the process, setting its state to P_ZOMBIE and exit_status to P_EXIT_SIGNALED. Signal 1 (P_SIGSTOP) suspends the process, removing it from ready queues. Signal 2 (P_SIGCONT) resumes a stopped process, returning it to the ready queue. The kernel function `k_signal_deliver()` handles the actual state transitions.
    - **Parameters**:
        - `pid` : The PID of the target process.
        - `signal` : The signal to send: 0 (P_SIGTERM), 1 (P_SIGSTOP), or 2 (P_SIGCONT).
    - **Return Value**: Returns `0` on success, or `-1` on error. `P_ERRNO` is set.
    - **Error Conditions**:
        - `P_ESRCH` if no process with the given `pid` exists
        - `P_EINVAL` if signal number is not 0, 1, or 2

- `void s_exit(void)`
    - **Description**: Terminates the calling process gracefully. Sets the process's exit status to P_EXIT_EXITED and changes state to P_ZOMBIE. Logs the exit event, removes the process from all queues, adopts any orphaned children to the init process, and exit the spthread. If the parent is blocked waiting (in s_waitpid), it will be unblocked. The process remains as a zombie until reaped by the parent. This function does not return.
    - **Return Value**: None (does not return).

- `int s_nice(pid_t pid, int priority)`
    - **Description**: Changes the scheduling priority of a process. Valid priorities are 0 (highest, interactive), 1 (normal), and 2 (lowest, batch). If the process is currently in the ready queue,

it is removed from its old priority queue and re-inserted into the new one. Priority changes are logged as NICE events.
    - **Parameters**:
        - `pid` : The PID of the target process.
        - `priority` : The new priority level (0, 1, or 2).
    - **Return Value**: Returns `0` on success, or `-1` on error. `P_ERRNO` is set.
    - **Error Conditions**:
        - `P_EINVAL` if priority is not in the range [0, 2]
        - `P_ESRCH` if no process with the given `pid` exists

- `void s_sleep(unsigned int ticks)`
    - **Description**: Blocks the calling process for a specified number of scheduler ticks. Sets the process's wake_tick to the target time and moves it to the blocked queue. The scheduler will automatically unblock the process when the wake time is reached. Uses a loop to prevent premature wakeup from signals (SIGSTOP/SIGCONT), ensuring the process only returns when the sleep duration has fully elapsed. If ticks is 0, returns immediately without blocking.
    - **Parameters**:
        - `ticks` : The number of clock ticks to sleep (0 for immediate return).
    - **Return Value**: None.
- `pid_t s_getpid(void)`
    - **Description**: Returns the process ID of the calling process.
    - **Return Value**: Returns the PID of the current process, or PID_INVALID (0) if no current process exists.
- `pcb_t** s_get_all_process(void)`
    - **Description**: Returns a pointer to the global process control block table, allowing access to all process structures in the system. The table is indexed by PID and contains pointers to PCB structures (or NULL for unused PIDs). This is primarily used by system utilities like `ps` to enumerate and display information about all running processes.
    - **Return Value**: Returns a pointer to the global `pcb_table` array of size MAX_PROC. Each entry is either a pointer to a `pcb_t` structure or NULL if that PID slot is not in use.
- `void s_shutdown(void)`
    - **Description**: Initiates a clean shutdown of the entire PennOS system by setting a global shutdown flag. The init process checks this flag in its main loop and terminates when shutdown is requested, causing all processes to exit. This is typically called by the `logout` command.
    - **Return Value**: None.