FileSystem.H:

Structures

- `fileptr_t`: Represents a file pointer, storing the process ID (`pid`) and the current pointer position (`ptr`) within a file.
- `file_t`: Represents file information, including filename, file ID, write access PID, a linked list of file pointers, and a pointer to the next file in the list.

File Operation Constants

- File Descriptor Constants: `F_STDIN`, `F_STDOUT`, `F_STDERR` are defined for standard input, output, and error.
- File Mode Constants: `F_WRITE`, `F_READ`, `F_APPEND` define different modes for opening files.
- File Seek Constants: `F_SEEK_SET`, `F_SEEK_CURR`, `F_SEEK_END` are used for positioning the file pointer.
- File Permission Constants: `FPERM_READ`, `FPERM_WRIT`, `FPERM_EXEC` represent read, write, and execute permissions.

Function Declarations

- `f_open`: Opens a file with a specified mode, returning a file descriptor.
- `f_read`: Reads data from a file descriptor into a buffer.
- `f_write`: Writes data to a file descriptor.
- `f_close`: Closes a file descriptor.
- `f_unlink`: Removes a file if it exists and is not in use.
- `f_lseek`: Repositions the file pointer within a file.
- `f_ls`: Lists a file or all files in the current directory.
- `f_touch`: Creates empty files with specified names.
- `f_print`: Prints a string to the terminal (stderr).
- `f_mount` and `f_unmount`: Mount and unmount the file system.
- `f_mv`: Renames or moves a file.
- `f_cp`: Copies a file.
- `f_rm`: Removes files.
- `f_chmod`: Changes file permissions.
- `print_fileptr_pids_all`: Debug function to print file pointer PIDs.
- `process_create_fileptrs` and `process_delete_fileptrs`: Functions for creating and deleting file pointers for a process.
- `find_file_entry_by_file_id`: Finds a file entry by its file ID.

FileSystem.C:

Overview

- The code is designed to interface with a file system, possibly a custom implementation like `pennfat`.
- It handles file operations such as opening, reading, writing, closing, and deleting files.
- There are also functions for moving, copying, and changing permissions of files, as well as listing file information.

Key Structures and Variables

- `file_t`: A structure representing an open file.
- `fileptr_t`: A structure representing a file pointer, with a process ID (`pid`) and a pointer/offset (`ptr`).
- `dir_entry_t`: Presumably a structure representing a directory entry in the file system.
- `open_files`: A global list of files currently open by any process.
- `next_file_id`: A global counter for the next available file ID for newly opened files.

Key Functions

File Management

`create_file_entry` and `delete_file_entry`: Functions to create and delete entries in the `open_files` list.
`create_fileptr` and `delete_fileptr`: Functions to manage file pointers, which keep track of the position within open files for each process.
`find_file_entry_by_file_id` and `find_file_entry_by_filename`: Functions to locate file entries by file ID or filename.
`f_open`: Opens or creates a file, returning a file descriptor.
`f_read`: Reads data from a file descriptor into a buffer.
`f_write`: Writes data to a file descriptor.
`f_close`: Closes a file descriptor, releasing resources.

`f_unlink`: Deletes a file from the file system.
`f_lseek`: Moves the file pointer to a specified position within a file.

Directory and File System Operations

`f_ls`: Lists information about files in the file system.
`f_touch`: Creates empty files with specified names.
`f_mount` and `f_unmount`: Functions to mount and unmount the file system.
`f_mv`: Moves or renames a file or directory.
`f_cp`: Copies a file or directory.
`f_rm`: Removes files or directories.
`f_chmod`: Changes permissions of a file or directory.

Utilities

`print_fileptr_pids_all`: Debugging function to print all file pointers and their associated process IDs.
`find_unused_fd`: Finds the first unused file descriptor in a process's file descriptor table.
`is_duplicate_fd`: Checks if a process already has a file descriptor open for a given file ID.
`f_isatty`: Checks whether a file descriptor refers to the terminal.
`f_print`: Prints a string to standard error.

Kernel Functions.c:

`k_process_create`

- Purpose: Creates a new PCB (Process Control Block) and adds it to the global PCB list.
- Parameters:
    - `parent`: A pointer to the parent PCB, if it exists.
- Functionality:
    - Calls `createPCB` to create a new PCB, passing the parent PCB if provided.
    - Adds the newly created PCB to the global PCB list using `addPCBToList`.
- Returns: The newly created PCB on success, or `NULL` on failure.

`k_process_kill`

- Purpose: Sends a specified signal to a given PCB.
- Parameters:
    - `process`: A pointer to the PCB to which the signal is to be sent.
    - `signal`: The signal to send.
- Functionality:
    - Logs the event of signaling a PCB using `log_signaled_event`.
    - Handles different signals:
        - `S_SIGTERM`: Marks the process as `T_ZOMBIED` (zombie state), logs it, and updates the state of its children to reflect orphan status.
        - `S_SIGSTOP`: Marks the process as `T_STOPPED` and logs the event.
        - `S_SIGCONT`: Marks the process as `T_RUNNING` and logs the event.
        - `S_SIGCHLD`: Continues the process by calling `k_process_kill` with `S_SIGCONT`.
- Returns: 0 on success; -1 on failure.

k_process_deep_cleanup

- Purpose: Frees the PCB and all its descendants.
- Parameters:
    - `process`: A pointer to the PCB to be freed, along with its descendants.
- Functionality:
    - Recursively frees all child PCBs using `k_free`.
    - Frees the specified PCB using `k_free`.

k_process_cleanup

- Purpose: Frees a single PCB.
- Parameters:
    - `process`: A pointer to the PCB to be freed.
- Functionality:
    - Sets the process status to `T_ZOMBIED`.
    - Removes the PCB from the global PCB list using `removePCBFromList`.

PCB.C:

get_tail

- Purpose: Retrieves the tail of a circular linked list.
- Parameters: `circular_ll` - pointer to the head of the circular linked list.

- Functionality: Iterates through the list until it finds the last element (where the next element is the head), indicating the tail of the circular list.
- Returns: The tail of the inputted linked list.

`k_free`

- Purpose: Frees an inputted PCB.
- Parameters: `process` - pointer to the PCB to be freed.
- Functionality: Frees the memory allocated for the PCB, including its context and stack space (using Valgrind stack deregistration if necessary).
- Returns: None.

`createPCB`

- Purpose: Creates a PCB, optionally inheriting properties from a parent.
- Parameters: `Parent` - pointer to the parent PCB, if it exists.
- Functionality: Allocates and initializes a new PCB, setting its PID, status, context, parent PID, priority, and file descriptors.
- Returns: The newly created PCB, or `NULL` on failure.

`addPCBToList`

- Purpose: Adds a given PCB to a circular linked list.
- Parameters: `head` - pointer to the head of the list; `pcb` - the PCB to add.
- Functionality: Adds the PCB to the circular linked list. If the list is empty, the PCB becomes the head; otherwise, it's added to the end of the list.
- Returns: None.

`removePCBFromList`

- Purpose: Removes a given PCB from a circular linked list.
- Parameters: `head` - pointer to the head of the list; `pcb` - the PCB to remove.
- Functionality: Removes the specified PCB from the list, updating the list's links appropriately.
- Returns: None.

`findPCBByPID`

- Purpose: Finds a PCB with a specific PID.
- Parameters: `pid` - the PID of the process to find.

- Functionality: Searches through the circular linked list for a PCB with the given PID.
- Returns: The PCB with the desired PID, if it exists.

`findPCBByContext`

- Purpose: Finds a PCB with a specific context.
- Parameters: `context` - the context to search for.
- Functionality: Searches through the circular linked list for a PCB with the given context.
- Returns: The PCB with the desired context, if it exists.

`getLength`

- Purpose: Gets the length of a circular linked list.
- Parameters: `head` - pointer to the head of the list.
- Functionality: Counts the number of elements in the list.
- Returns: The length of the PCB list.

`count_running`

- Purpose: Counts the number of `T_RUNNING` processes in a circular linked list.
- Parameters: `head` - pointer to the head of the list.
- Functionality: Iterates through the list, counting processes with the status `T_RUNNING`.
- Returns: The number of `T_RUNNING` processes.

`count_running_priority`

- Purpose: Counts the number of `T_RUNNING` processes with a specific priority.
- Parameters: `head` - pointer to the head of the list; `prio` - the desired priority.
- Functionality: Iterates through the list, counting processes with the status `T_RUNNING` and the specified priority.
- Returns: The number of relevant processes.

PCB.H:

PCB Structure Definition

- `PCB` Structure: Represents a process control block in the system.

- name: A string representing the name of the process.
- context: A pointer to the ucontext_t structure, which holds the CPU context for the process.
- parent_pid: PID of the parent process.
- pid: The process ID.
- children: An array to store the PIDs of child processes.
- numChildren: The number of children processes.
- fileDescriptors: An array of file descriptors used by the process.
- priority: The priority of the process.
- status: The current status of the process (e.g., running, stopped).
- next: A pointer to the next PCB in a linked list.

Constants

- STACKSIZE: Defines the size of the stack for each process. It's currently set to 4096 * 256 bytes.
- File Descriptor Constants: NOFILE, STDIN_ID, STDOUT_ID, and STDERR_ID are constants defining special file descriptor values.

Global Variables

- pcb_list: A global pointer to the head of the PCB list.
- next_pid: A global variable to keep track of the next available PID.

P-User Functions.C:

p_spawn

- Purpose: Spawns a new process (PCB) with a specified start function, arguments, and I/O file descriptors.
- Functionality: Creates a child process, sets its file descriptors for standard input/output, and initializes its context and stack.
- Returns: PID of the spawned PCB on success; -1 on failure.

p_waitpid

- Purpose: Waits for a specified process to change its state, with an option to return immediately.
- Functionality: If nohang is false, it blocks until the specified process changes its state. If nohang is true, it returns immediately.
- Returns: 0 on success, -1 on failure.

`p_kill`

- Purpose: Sends a signal to a process with a specified PID.
- Functionality: Deletes file pointers associated with the process and sends the specified signal.
- Returns: 0 on success; -1 on failure.

`p_nice`

- Purpose: Changes the priority of a specified process.
- Functionality: Finds the process by PID and changes its priority, logging the event.
- Returns: 0 on success; -1 on failure.

`p_sleep`

- Purpose: Blocks the current process for a specified number of ticks.
- Functionality: Blocks the calling process for the specified time, logging block and unblock events.

`p_exit`

- Purpose: Exits the current process unconditionally.
- Functionality: Sets the current process status to `T_ZOMBIED` (zombie state), handling child processes and logging the event.

Status Check Functions

- `W_WIFEXITED`, `W_WIFSTOPPED`, `W_WIFCONTINUED`, `W_WIFSIGNALED`: These functions return boolean values indicating whether a child process exited normally, was stopped, continued, or terminated by a signal, respectively.

Scheduler.C:

Scheduler and Context Management

- `scheduler`: The main scheduling function. It decides which PCB to run for the current tick based on a priority system. The scheduler employs a form of priority scheduling with a random element (`roulette`), determining which priority level to serve next.
- `reaper`: A cleanup function that increments the global tick count and returns control to the scheduler. It's meant to run when a process terminates.
- `alarmHandler`: This function is triggered by a timer signal (SIGALRM) and is responsible for time slicing. It increments the tick count and triggers a context switch to the scheduler.
- `setAlarmHandler` and `setTimer`: These functions set up the timer and alarm handler to enable time slicing in the scheduler.
- `freeStacks`: Frees all allocated stacks before exiting the program. This function is commented out, indicating it's either not implemented or not currently used.

Scheduler Initialization and Start

- `start_scheduler`: Initializes the scheduler and its context. It sets up signal handling to ignore certain signals (like Ctrl-C, Ctrl-, and Ctrl-Z) and creates contexts for the scheduler and the reaper function. It also registers the stacks with Valgrind (for memory debugging) and starts the scheduler.

Global Variables and Constants

- `mainContext`, `schedulerContext`, `reaperContext`: These are the ucontext_t variables used to store the contexts for the main program, scheduler, and reaper function.
- `activeContext`: A pointer to the currently active context.
- `centisecond`: A constant representing 10 milliseconds, used in the timer setup.

Functionality

- The scheduler function uses a random selection method based on priorities to decide which PCB to execute next.

- The system uses UNIX signals and `ucontext` for preemptive multitasking, where each process (PCB) is given a time slice to run.
- The alarm handler, invoked every 10 milliseconds, ensures regular context switching, simulating a preemptive multitasking environment.

Logger.C:

Global Variable

- `logfile`: A `FILE*` pointer to the log file where events are recorded.

Logging Functions

Each function follows a similar pattern: writing a formatted log message to `logfile` with relevant process information. The functions and their purposes are:

`log_schedule_event`: Logs a scheduling event, indicating when a process is scheduled to run.
`log_create_event`: Logs a process creation event.
`log_signaled_event`: Logs a signaling event, such as when a process receives a signal.
`log_exited_event`: Logs a process exit event, specifically a natural termination.
`log_zombie_event`: Logs when a process becomes a zombie.
`log_orphan_event`: Logs when a process becomes an orphan.
`log_waited_event`: Logs when a process enters a waiting state.
`log_nice_event`: Logs a priority change event for a process.
`log_blocked_event`: Logs when a process is blocked.
`log_unblocked_event`: Logs when a process is unblocked.
`log_stopped_event`: Logs when a process is stopped, typically by a signal.
`log_continued_event`: Logs when a process is continued from a stopped state.

Format of Log Entries

Each log entry includes:

- The global tick count (`ticks`) at the time of the event.
- The process ID (`pid`).
- The process's priority (`prio`).

- The name of the process (`process_name`).
- The type of event (e.g., `SCHEDULE`, `CREATE`, `EXITED`).

Functionality and Design

- The code provides a standardized way to record process-related events, which is crucial for debugging and monitoring in an operating system.
- Each logging function takes specific details about the event and process, ensuring that the logs contain relevant and detailed information.

FAT.H

Fat.c:

FAT File System Operations

- `mem_idx`: Calculates the memory address of a specific block index in the file system.
- `get_free_block`: Searches for an open block in the file system.
- `delete_chain`: Deletes a chain of blocks in the file system, marking them as free.
- `build_chain`: Allocates and fills a chain of data blocks in the file system.
- `fill_chain`: Fills a FAT chain with data from a buffer.
- `add_file`: Adds a new empty file to a directory, allocating new blocks as necessary.
- `write_file`: Writes a file block to memory.
- `read_chain`: Reads data from a FAT chain into a buffer.
- `find_file`: Finds a file or directory in the file system.
- `valid_filename`: Checks if a string is a valid filename.
- `fs_getmeta`: Retrieves metadata information from the file system.
- `fs_mount`: Mounts a file system.
- `fs_unmount`: Unmounts a file system.
- `fs_touch`: Creates or updates a file in the file system.
- `fs_mv`: Moves or renames a file in the file system.
- `fs_mark_deleted`: Marks a file or directory as deleted.
- `fs_rm`: Removes a file or directory.
- `fs_cat`: Concatenates input strings or files and outputs the result.
- `fs_cp`: Copies a file or directory.
- `fs_ls_single`: Displays information about a single directory entry.
- `fs_ls`: Displays information about all directory entries in the file system.
- `fs_chmod`: Changes the permissions of a file or directory.

Safe.C:

File Operations

`safe_open`: Opens a file and returns the file descriptor. If it fails, an error message is printed and the program exits.
`safe_close`: Closes a file descriptor. If it fails, an error message is printed and the program exits.

`safe_read`: Reads data from a file descriptor into a buffer. If it fails, an error message is printed and the program exits.
`safe_write`: Writes data from a buffer to a file descriptor. If it fails, an error message is printed and the program exits.
`safe_lseek`: Performs a seek operation on a file descriptor. If it fails, an error message is printed and the program exits.

Memory Mapping Operations

`safe_msync`: Synchronizes a file mapping with the physical storage. If it fails, an error message is printed and the program exits.
`safe_mmap`: Maps files or devices into memory. If it fails, an error message is printed and the program exits.
`safe_munmap`: Unmaps files or devices from memory. If it fails, an error message is printed and the program exits.

Job-List.C:
Each job in the list is represented by a `job_t` structure, which holds information about the job's ID, process ID, stop order, completion status, and a pointer to the next job in the list. Let's examine each function:

Job List Management Functions

`job_find_by_jobid`: Searches for a job in the linked list by its job ID and returns a pointer to the found job or `NULL` if not found.
`job_get_last`: Retrieves the job ID of the last job in the linked list, or returns `-1` if the list is empty.
`jobs_push`: Adds a new job to the end of the linked list. It allocates memory for the new job, sets its details, and returns a pointer to it.
`jobs_insert`: Inserts a new job into the list in ascending order based on the job ID.
`jobs_remove`: Removes a job from the linked list by its job ID and returns a pointer to the removed job. If the job is not found, it returns `NULL`.
`job_print`: Prints information about a job, including its ID, process ID, and status. It uses `p_waitpid` to get the job's current status and then formats this information into a buffer for printing.

Pennoshell.C:

Core Shell Functionality

Command Handling: The shell supports various commands, including `cat`, `sleep`, `echo`, `ls`, `touch`, `mv`, `cp`, `rm`, `chmod`, `ps`, `kill`, `zombify`, `orphanify`, etc. Each command has a corresponding function (`shell_<command>`) that is called when the command is entered.
Job Control: Jobs (processes) can be managed using commands like `bg`, `fg`, `jobs`, and `logout`. Jobs can be moved between foreground and background, and their statuses are tracked.
Signal Handling: Custom signal handlers (`stop_handler` and `term_handler`) are implemented for `SIGTSTP` and `SIGINT` to manage process interruptions and terminations.
Input Parsing and Execution: The shell parses input commands using `parse_command` and executes them accordingly. It supports standard I/O redirection and background execution (`&`).
Script Execution: The shell can execute scripts (simple sequences of commands) with the `execute_script` function.
Foreground and Background Job Management: The shell tracks foreground and background jobs, managing them with a linked list (`job_t*`).

Specific Functions

`spawn_command` and `execute_command`: These functions handle the spawning and execution of individual commands, setting up file descriptors for input and output redirection.
`cull_background` and `empty_reaped`: These functions manage background jobs, culling finished jobs, and handling zombie processes.
Job Control Functions (`bg`, `fg`, `jobs`): These functions allow the user to control job execution, including moving jobs between foreground and background and listing all jobs.
Signal Handling (`stop_handler` and `term_handler`): These handlers manage signals received by the shell, particularly for stopping and terminating foreground jobs.
`pennos_shell` Main Loop: The main loop of the shell (`pennos_shell` function) prompts for input, parses commands, and manages the execution flow. It handles all functionalities and manages the job control logic.

Pennfat.c:

Core CLI Functionality

Filesystem Management: The CLI can create (`mkfs`), mount (`mount`), and unmount (`unmount`) filesystems. It uses a FAT to manage file locations.
File Operations: Commands like `touch`, `mv`, `rm`, `cat`, `cp`, and `chmod` are used for creating, moving, removing, reading, copying, and changing file permissions, respectively.
Input Parsing and Execution: The CLI reads commands from the standard input, parses them, and executes the corresponding filesystem operations.
Hex Dump (`hd`): This command displays the filesystem content in hexadecimal format, with options to show ASCII characters (`-c`), limit the number of bytes displayed (`-n`), and display block numbers (`-b`).

Specific Functions

Filesystem and File Validation: Functions like `valid_fs_mounted` and `all_files_exist` check the state of the filesystem and the existence of files, ensuring the commands are executed in a valid context.
Utility Functions: `correct_argc` ensures the right number of arguments for a command, improving the robustness of command processing.
FAT Management: The CLI interacts with the FAT for file location management, using functions like `fs_touch`, `fs_mv`, `fs_rm`, etc.
Safe Wrappers: Functions prefixed with `safe_` (e.g., `safe_open`, `safe_read`) are used for error-handled versions of standard system calls.

Pennos.C

Entry point for PennOS.
Initializes the logger, filesystem, and spawns the shell process.
@param argc The number of command-line arguments.
@param argv An array of command-line arguments.
@return Returns 1 if the number of command-line arguments is less than 2.

Utils Functioin:

`p-errno.h` and Error Handling

- Error Codes: Defines a series of error codes (`ERR_NONE`, `ERR_FS_FILE_NOT_FOUND`, etc.) for various failure scenarios.
- Error Strings: Provides `err_string` function to convert error codes to human-readable descriptions.
- Custom Error Printing: Implements `p_perror` function to display errors, combining a custom message with the string representation of the current `ERRNO`.

`safe-user.h` - Safe Wrappers for System Calls

- Implements a set of "safe" wrapper functions (`safe_f_open`, `safe_f_read`, etc.) for various file and process-related operations (`f_open`, `f_read`, `f_write`, `f_close`, `f_lseek`, `f_unlink`, `f_print`, `p_spawn`, `p_waitpid`, `p_kill`, `p_nice`).
- These functions call the original system calls and check for errors. If an error is detected, they print an error message using `p_perror` and terminate the process using `p_exit`.

`util.h` - General Utility Functions

- Buffer Sizes: Defines constants for input/output buffer sizes (`IOBUFFER_SIZE`) and error message buffer sizes (`ERRBUFFER_SIZE`).
- Argument Count: Provides `get_argc` to count the number of arguments in a string array.
- Memory Allocation: Includes `safe_malloc` for safely allocating memory, with error checking.
- Signal Handling: Implements `safe_signal` to set signal handlers with error checking.