Linux-style Tree of Files

```
|-- Makefile
|-- bin
|   |-- pennfat
|   `-- pennos
|-- doc
|   |-- README.md
|   `-- CompanionDocument.pdf
|-- log
|   |-- log
`-- src
    |-- pennfat.c
    |-- pennos.c
    `-- util
        `-- spthread.c
        `-- spthread.h
        `-- builtins.h
        `-- builtins.c
        `-- globals.h
        `-- macros.h
        `-- os_errors.h
        `-- os_errors.c
        `-- parser.h
        `-- PCB.h
        `-- PCBDeque.c
        `-- PCBDeque.h
        `-- PIDDeque.c
        `-- PIDDeque.h
        `-- terminal_history.c
        `-- terminal_history.h

    `-- fat
        `-- fat_globals.h
        `-- fat_helper.c
        `-- fat_helper.h
    `-- kernel
        `-- kernel.h
        `-- kernel.c
        `-- kernel_system.h
        `-- kernel_system.c
        `-- shell.c
        `-- shell.h
        `-- stress.h
        `-- stress.c
```

PCB Struct

The PCB struct has the following fields:

- **pid_t pid**: the process id, incremented each time a new process is spawned
- **int status**: a code for the status of the process
- **pid_t parent_pid**: parent process id, -1 if no parent
- **spthread_t curr_thread**: the thread that this PCB is running
- **PIDDeque* child_pids**: a list of the PIDs of this process' children
- **PIDDeque* status_changes**: a list of all of the PIDs that have seen their status update
- **int blocking**: 1 if blocking, 0 if not
- **int priority**: priority level between 0 and 2
- **int sleep_duration**: number of quanta to sleep for. If not sleeping, set sleep_duration = -1;
- **char* process_name**: name of process
- **int stop_time**: when it was stopped
- **bool is_background**: is it in the background
- **int process_fdt[1024]**: process-level file descriptor table
- **struct parsed_command* parsed**: the command corresponding to this process
- **int job_id**: used for storing JobID

Data Structures

All of our data structures are initialized in PennOS.c

We have PCBList, which is of type PCBDeque*. This will store all of the processes that have not yet been reaped. This is the source of truth for currently active processes, and is essential for storing all state throughout the OS.

We have a priorityList, which is of type PIDDeque[4]. This is an array of 4 PIDDeques, representing priority levels 0-2 as well as stopped/blocked/inactive jobs. We use this to keep track of the status of processes, and to select them for scheduling. We have an algorithm in the scheduler that picks a certain priority level, and then we schedule the first job in the corresponding PIDDeque.

Signals

We define only three signals: P_SIGSTOP, P_SIGCONT, and P_SIGTERM. We give them integer codes 69, 70, and 71. P_SIGSTOP represents the action of stopping a job, such as by Ctrl-Z. P_SIGCONT is used to continue stopped jobs. P_SIGTERM is meant to kill or terminate jobs.

Penn FAT

fat_helper.h

```
/**
* @brief Mounts the file system
*
* @param fs_name Name of the file system to mount
* @param num_blocks Number of blocks in the file system
* @param block_size Size of each block in the file system
* @return int 0 if successful, -1 if error
*/
int mount(char* fs_name, int* num_blocks, int* block_size);
```

```
/**
* @brief Finds permissions for a file
*
* @param file_name Name of the file to find permissions for
* @return int Permission of the file if successful, -1 if error
*/
int k_findperm(char* file_name);
```

```
/**
* @brief Creates the files if they do not exist, or updates their timestamp to
* the current system time
*
* @param file_name File to create/update
* @return int 0 if successful, -1 if error
*/
int k_touch(char* file_name);
```

```
/**
* @brief Renames the source file to the destination file
*
* @param source_file File to rename
* @param dest_file New name of the file
* @return int 0 if successful, -1 if error
*/
int k_mv(char* source_file, char* dest_file);
```

```
/**
* @brief Changes permissions for a file
```

```
*
* @param file_name File to change permissions for
* @param perm Permissions to change to
* @param modify Either '+' or '-', corresponding to adding or removing
* permissions
* @return int 0 if successful, -1 if error
*/
int k_chmod(char* file_name, int perm, char modify);
```

```
/**
* @brief List all files in the directory
*
* @param output_fd File to write the output to (STDOUT if NULL)
*
* @return int 0 if successful, -1 if error
*/
int k_ls_all(int output_fd);
```

```
/**
* @brief Check if a file exists in the directory
*
* @param file_name File to check
* @param directory [Output Parameter] Pointer to the directory entry of the
* file
* @param location [Output Parameter] Pointer to the location entry of the file
* @return int 0 if file does not exist, 1 if file exists, 2 if error
*/
int k_file_exists(char* file_name, dir_entry* directory, loc_entry* location);
```

```
/**
* @brief Retrieve the metadata of the file system
*
* @param num_blocks [Output Parameter] Pointer to the number of blocks in the
* file system
* @param block_size [Output Parameter] Pointer to the size of each block in the
* file system
*/
void k_metadata(int* num_blocks, int* block_size);
```

```
/**
* @brief Find the first open entry in the FAT
```

```
*
* @return int -1 if no open entries, otherwise the index of the first open
* entry
*/
int k_open_entry();
```

```
/**
* @brief Updates size of file in directory entry
*
* @param file_name File to update size for
* @param new_size Size to update to
* @return int 0 if successful, -1 if error
*/
int update_file_size(char* file_name, uint32_t new_size);
```

```
/**
* @brief Updates the size of a file in the Global File Descriptor Table
*
* @param file_name File to update size for
* @param new_size Size to update to
* @return int 0 if successful, -1 if error
*/
int update_size_fdt(char* file_name, uint32_t new_size);
```

```
/**
* @brief Updates the offset of a file in the Global File Descriptor Table
*
* @param file_name File to update offset for
* @param new_offset Offset to update to
* @return int 0 if successful, -1 if error
*/
int update_offset_fdt(char* file_name, uint32_t new_offset);
```

```
/**
* @brief Allocates new blocks in the FAT for a file if needed
```

```
*
* @param file_name File to update size for
* @param new_size Size to update to
* @return int 0 if successful, -1 if error
*/
int update_file_size(char* file_name, uint32_t new_size);
```

```
/**
* @brief Removes a file from the file system
*
* @param fName File to remove
* @return int 0 if successful, -1 if error
*/
int k_unlink(const char* fname);
```

```
/**
* @brief List the filename/filenames in the current directory
*
* @param filename File to list (NULL if all files)
* @param output_fd File to write the output to (STDOUT if NULL)
* @return int 0 if successful, 1 if error
*/
int k_ls(const char* filename, int output_fd);
```

```
/**
* @brief Close the file indicated by fd
*
* @param fd File descriptor to close
* @return int 0 if successful, -1 if error
*/
int k_close(int fd);
```

```
/**
* @brief Repositions the file pointer for file indicated by fd to the offset
```

```
* relative to whence
*
* @param fd File descriptor to reposition
* @param offset Number of bytes to move the file pointer
* @param whence F_SEEK_SET, F_SEEK_CUR, or F_SEEK_END
* @return int 0 if successful, -1 if error
*/
int k_lseek(int fd, int offset, int whence);
```

```
/**
* @brief Write n bytes of the string referenced by str to the file fd and
* increment the file pointer by n
*
* @param fd File descriptor to write to
* @param str String to write
* @param n Number of bytes to write
* @return int Number of bytes written, -1 if error
*/
int k_write(int fd, const char* str, int n);
```

```
/**
* @brief Read n bytes from the file referenced by fd
*
* @param fd File descriptor to read from
* @param n Number of bytes to read
* @param buf Buffer to store the read bytes
* @return int Number of bytes read, 0 if EOF reached, -1 if error
*/
int k_read(int fd, int n, char* buf);
```

```
/**
* @brief Opens a file with the given name and mode
*
* @param fName File to open
* @param mode F_READ, F_WRITE, or F_APPEND
* @return int File descriptor if successful, -1 if error
*/
int k_open(const char* fName, int mode);
```

Kernel

kernel_system.h

```
/**
* @brief Create a child process that executes the function `func`.
* The child will retain some attributes of the parent.
*
* @param func Function to be executed by the child process.
* @param argv Null-terminated array of args, including the command name as
* argv[0].
* @param fd0 Input file descriptor.
* @param fd1 Output file descriptor.
* @param process_name Name of the process.
* @return pid_t The process ID of the created child process.
*/
pid_t s_spawn(void* (*func)(void*),
              char* argv[],
              int fd0,
              int fd1,
              char* process_name,
              bool is_background,
              struct parsed_command* parsed);
```

```
/**
* @brief 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.
*
* @param pid Process ID of the child to wait for.
* @param wstatus Pointer to an integer variable where the status will be
* stored.
* @param nohang If true, return immediately if no child has exited.
* @return pid_t The process ID of the child which has changed state on success,
* -1 on error.
*/
pid_t s_waitpid(pid_t pid, int* wstatus, bool nohang);
```

```c
/**
* @brief Send a signal to a particular process.
*
* @param pid Process ID of the target proces.
* @param signal Signal number to be sent.
* @return 0 on success, -1 on error.
*/
int s_kill(pid_t pid, int signal);
```

```c
/**
* @brief Unconditionally exit the calling process.
*/
void s_exit(void);
```

```c
/**
* @brief Set the priority of the specified thread.
*
* @param pid Process ID of the target thread.
* @param priority The new priorty value of the thread (0, 1, or 2)
* @return 0 on success, -1 on failure.
*/
int s_nice(pid_t pid, int priority);
```

```c
/**
* @brief 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.
*
* @param ticks Duration of the sleep in system clock ticks. Must be greater
* than 0.
*/
void s_sleep(unsigned int ticks);
```

```c
/**
* @brief Write a message to the system log.
*
* @param message Message to be written to the log.
*/
void s_log(char* message);
```

```c
/**
* @brief Prints information about all processes on the system.
*
*/
void s_ps();
```

```c
/**
* @brief Creates the files if they do not exist, or updates their timestamp to
* the current system time
*
* @param file_name File to create/update
* @return int 0 if successful, -1 if error
*/
int s_touch(char* fname);
```

```c
/**
* @brief Renames the source file to the destination file
*
* @param source_file File to rename
* @param dest_file New name of the file
* @return int 0 if successful, -1 if error
*/
int s_mv(char* source_file, char* dest_file);
```

```c
/**
* @brief Changes permissions for a file
*
* @param file_name File to change permissions for
* @param perm Permissions to change to
* @param modify Either '+' or '-', corresponding to adding or removing
* permissions
* @return int 0 if successful, -1 if error
*/
int s_chmod(char* file_name, int perm, char modify);
```

```
/**
* @brief Opens a file with the given name and mode
*
* @param fName File to open
* @param mode F_READ, F_WRITE, or F_APPEND
* @return int File descriptor if successful, -1 if error
*/
int s_open(const char* fname, int mode);
```

```
/**
* @brief Read n bytes from the file referenced by fd
*
* @param fd File descriptor to read from
* @param n Number of bytes to read
* @param buf Buffer to store the read bytes
* @return int Number of bytes read, 0 if EOF reached, -1 if error
*/
int s_read(int fd, int n, char* buf);
```

```
/**
* @brief Write n bytes of the string referenced by str to the file fd and
* increment the file pointer by n
*
* @param fd File descriptor to write to
* @param str String to write
* @param n Number of bytes to write
* @return int Number of bytes written, -1 if error
*/
int s_write(int fd, const char* str, int n);
```

```
/**
* @brief Close the file indicated by fd
*
* @param fd File descriptor to close
* @return int 0 if successful, -1 if error
*/
int s_close(int fd);
```

```
/**
* @brief Removes a file from the file system
*
* @param fName File to remove
* @return int 0 if successful, -1 if error
*/
int s_unlink(const char* fname);
```

```
/**
* @brief Repositions the file pointer for file indicated by fd to the offset
* relative to whence
*
* @param fd File descriptor to reposition
* @param offset Number of bytes to move the file pointer
* @param whence F_SEEK_SET, F_SEEK_CUR, or F_SEEK_END
* @return int 0 if successful, -1 if error
*/
int s_lseek(int fd, int offset, int whence);
```

```
/**
* @brief List the filename/filenames in the current directory
*
* @param filename File to list (NULL if all files)
* @param output_fd File to write the output to (STDOUT if NULL)
* @return int 0 if successful, -1 if error
*/
int s_ls(const char* filename, int output_fd);
```

```
/**
* @brief Returns the permission fo the filename
*
* @param filename File to get permission
* @return int representing the permission of the file
*/
int s_findperm(char* filename);
```

```
/**
* @brief Handles the bg command on specfied pid
*
* @param pid job to resume in the background, -1 if no job provided to bg
* command
* @return 0 on successs, -1 on error
*/
int s_handle_bg(pid_t pid);
```

```
/**
* @brief Handles thefbg command on specfied pid
*
* @param pid job to resume in the foreground, -1 if no job provided to fg
* command
* @return 0 on successs, -1 on error
*/
int s_handle_fg(pid_t pid);
```

kernel.h

```
/**
* @brief Create a new child process, inheriting applicable properties from the
* parent.
*
* @return Reference to the child PCB.
*/
pcb* k_proc_create(pcb* parent,
                   spthread_t thread,
                   int fd0,
                   int fd1,
                   char* process_name,
                   bool is_background,
                   struct parsed_command* parsed);
```

```
/**
* @brief Get the PCB of the currently running process.
*
* @return Reference to the child PCB.
*/
pcb* k_get_proc(void);
```

```
/**
* @brief Sends a signal to a PID
*
* @return Returns 0 on success, -1 on failure.
*/
int k_send_signal(pid_t pid, int signal);
```

```
/**
* @brief Change priority of a pid to given priority
*
* @return Returns 0 on success, -1 on failure.
*/
int k_change_priority(pid_t pid, int priority);
```

```
/**
* @brief Wait on child of the calling process.
*
* @return Returns pid_of child, -1 on failure.
*/
pid_t k_waitpid(pid_t pid, int* wstatus, bool nohang);
```

```c
/**
* @brief Exits out of the calling process. Doesn't clean up the process
*
* @return nothing
*/
void k_exit(void);
```

```c
/**
* @brief Sleeps for ticks amount of time
*
* @return nothing
*/
void k_sleep(unsigned int ticks);
```

```c
/**
* @brief Clean up a terminated/finished thread's resources.
* This may include freeing the PCB, handling children, etc.
*/
void k_proc_cleanup(pcb* proc);
```

```c
/**
* @brief Checks for all sleeping and running jobs. Maintains a counter for how
* long they have been sleeping for. Once a slept job has finished running,
* change the status and informs the parent, scheduling the parent if necessary.
* @return nothing
*/
void k_sleep_check(void);
```

```c
/**
* @brief Helper function called within waitpid, used to handle a status change
* of the job specified in target_pid, calls k_proc_cleanup if necessary.
* @return nothing
*/
void k_handle_status_changes(pid_t target_pid);
```

```c
/**
* @brief Function which writes status updates to the log file.
* @return nothing
*/
void k_write_log(char* message);
```

```
/**
* @brief Kernel function which handles the ps job command.
* @return nothing
*/
void k_ps();
```

```
/**
* @brief Helper function which maps the status of a job to a char used for
* logging and job status updates
* @return nothing
*/
char* get_status(int status);
```

```
/**
* @brief Function which handles the 'bg' command on the specified pid. If no
* pid is provided to bg, the input to k_handle_bg is -1.
* @return 0 on success, -1 on error
*/
int k_handle_bg(pid_t pid);
```

```
/**
* @brief Function which handles the 'fg' command on the specified pid. If no
* pid is provided to fg, the input to k_handle_fg is -1.
* @return 0 on success, -1 on error
*/
int k_handle_fg(pid_t pid);
```

Shell

shell.h

```
/**
* @brief Manages the input/output redirection for a command.
*
* @param parsed Parsed command struct to check for redirection
* @param in_file Pointer to the file descriptor for input
* @param out_file Pointer to the file descriptor for output
* @return true If the redirection was successful
* @return false If the redirection was unsuccessful
*/
bool handle_io_setup(struct parsed_command* parsed,
                     int* in_file,
                     int* out_file);
```

```
/**
* @brief Matches a command to a builtin function.
*
* @param str String to match
* @param os_proc_func Function pointer to the matched function
* @param process_name Name of the matched function
*/
void builtin_matcher(char* str,
                     void* (**os_proc_func)(void*),
                     char** process_name);
```

```
/**
* @brief Prints out shell prompt and reads user input into a buffer
*
* @param cmd
* @param read_res Pointer to the result of the read indicating the number of
* bytes read
*/
void read_command(char* cmd, ssize_t* read_res);
```

```
/**
* @brief Function to be executed by the shell thread.
*
*/
void*(shell)(void*);
```

Utility

Builtins.h

```
/**
* @brief Counts the number of arguments in a command
*
* @param args NULL-terminated list of arguments
* @return int Number of arguments in the list
*/
int num_arg(char** args);
```

```
/**
* @brief The ususal `cat` program.
*
* If `files` arg is provided, concatenate these files and print to stdout
* If `files` arg is *not* provided, read from stdin and print back to stdout
*
* Example Usage: cat f1 f2 (concatenates f1 and f2 and print to stdout)
* Example Usage: cat f1 f2 < f3 (concatenates f1 and f2 and prints to stdout,
* ignores f3)
* Example Usage: cat < f3 (concatenates f3, prints to stdout)
*/
void* cat(void* arg);
```

```
/**
* @brief Sleep for `n` seconds.
*
*
* Example Usage: sleep 10
*/
void* os_sleep(void* arg);
```

```
/**
* @brief Busy wait indefinitely.
* It can only be interrupted via signals.
*
* Example Usage: busy
*/
void* busy(void* arg);
```

```
/**
```

```
* @brief Echo back an input string.
*
* Example Usage: echo Hello World
*/
void* echo(void* arg);
```

```
/**
* @brief Lists all files in the working directory.
*
* Example Usage: ls (regular credit)
* Example Usage: ls ../../foo/./bar/sample (only for EC)
*/
void* ls(void* arg);
```

```
/**
* @brief For each file, create an empty file if it doesn't exist, else update
* its timestamp.
*
* Example Usage: touch f1 f2 f3 f4 f5
*/
void* touch(void* arg);
```

```
/**
* @brief 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
*/
void* mv(void* arg);
```

```
/**
* 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
*/
void* cp(void* arg);
```

```
/**
* @brief 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
*/
void* rm(void* arg);
```

```
/**
* @brief 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)
*/
void* chmod(void* arg);
```

```
/**
* @brief List all processes on PennOS, displaying PID, PPID, priority, status,
* and command name.
*
* Example Usage: ps
*/
void* ps(void* arg);
```

```
/**
* @brief 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)
*/
void* os_kill(void* arg);
```

```
/**
* @brief 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)
*/
void* fg(void* arg);
```

```
/**
* @brief 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)
*/
void* bg(void* arg);
```

```
/**
* @brief Spawn a new process for `command` and set its priority to `priority`.
* 2. Adjust the priority level of an existing process.
*
* Example Usage: nice 2 cat f1 f2 f3 (spawns cat with priority 2)
*/
void* u_nice(void* arg);
```

```
/**
* @brief Adjust the priority level of an existing process.
*
* Example Usage: nice_pid 0 123 (sets priority 0 to PID 123)
*/
void* nice_pid(void* arg);
```

```
/**
* @brief Helper for zombify.
*/
void* zombie_child(void* arg);
```

```
/**
* @brief Used to test zombifying functionality of your kernel.
*
* Example Usage: zombify
*/
void* zombify(void* arg);
```

```
/**
* @brief Helper for orphanify.
*/
void* orphan_child(void* arg);
```

```
/**
* @brief Used to test orphanifying functionality of your kernel.
*
* Example Usage: orphanify
*/
void* orphanify(void* arg);
```

```
/**
```

```
* @brief Lists all jobs.
*
* Example Usage: jobs
*/
void* jobs(void* arg);
```

```
/**
* @brief Lists all available commands.
*
* Example Usage: man
*/
void* man(void* arg);
```

```
/**
* @brief Exits the shell and shutsdown PennOS.
*
* Example Usage: logout
*/
void* logout(void* arg);
```

PCBDeque.h

```
// A single node within a deque.
//
// A node contains next and prev pointers as well as a pointer to a PCB struct.
typedef struct pcb_dq_node {
 pcb* pcb;                  // Stores PCB
 struct pcb_dq_node* next;  // next node in deque, or NULL
 struct pcb_dq_node* prev;  // prev node in deque, or NULL
} PCBDqNode;

// The entire Deque.
// This struct contains metadata about the deque.
typedef struct dq_st {
 int num_elements;  //  # elements in the list
 PCBDqNode* front;  // beginning of deque, or NULL if empty
 PCBDqNode* back;   // end of deque, or NULL if empty
} PCBDeque;

// !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
// "Methods" for our Deque implementation.
// !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

/**
* @brief Allocates and returns a pointer to a new Deque.
*
* @return pointer to the deque, NULL on error
*/
PCBDeque* PCBDeque_Allocate(void);

/**
* @brief Free a Deque that was previously allocated by Deque_Alocate
*
* @param deque: the deque pointer to free. Will be unsafe to use after.
*/
void PCBDeque_Free(PCBDeque* deque);

/**
* @brief Return the number of elements in the Deque.
*
* @param deque: the pointer to the deque we are measuring
* @return pointer to the deque, NULL on error
```

```
*/
int PCBDeque_Size(PCBDeque* deque);

/**
* @brief Add a new element to the front of the Deque.
*
* @param deque: the pointer to the Deque we are adding to
* @param payload: the pointer to the struct we are adding
*/
void PCBDeque_Push_Front(PCBDeque* deque, pcb* payload);

/**
* @brief Pop an element from the front of the Deque.
*
* @param deque: the deque that we are popping from
* @return true on success, false if the deque is empty or error
*/
bool PCBDeque_Pop_Front(PCBDeque* deque);

/**
* @brief Peek at the element at the front of the deque.
*
* @param deque: the deque we are peeking at
* @param payload_ptr: a return parameter; a pointer that will point to the
* peeked struct
* @return true on success, false if the deque is empty or error
*/
bool PCBDeque_Peek_Front(PCBDeque* deque, pcb* payload_ptr);

/**
* @brief Add a new element to the back of the Deque.
*
* @param deque: the pointer to the Deque we are adding to
* @param payload: the pointer to the struct we are adding
*/
void PCBDeque_Push_Back(PCBDeque* deque, pcb* payload);

/**
* @brief Pop an element from the back of the Deque.
*
* @param deque: the deque that we are popping from
* @return true on success, false if the deque is empty or error
```

```
*/
bool PCBDeque_Pop_Back(PCBDeque* deque);

/**
* @brief Peek at the element at the back of the deque.
*
* @param deque: the deque we are peeking at
* @param payload_ptr: a return parameter; a pointer that will point to the
* peeked struct
* @return true on success, false if the deque is empty or error
*/
bool PCBDeque_Peek_Back(PCBDeque* deque, pcb** payload_ptr);

/**
* @brief Search the Deque from a struct containing a certain process/jobID
*
* @param deque: the deque to search inside
* @param job_id: the pid that we are looking for
* @return A pointer to an element in the deque that matches; NULL if no match
* found
*/
pcb* PCBDequeJobSearch(PCBDeque* deque, int job_id);

/**
* @brief Search the Deque from a struct containing a certain process/jobID and
* delete it
*
* @param deque: the deque to search inside
* @param pgid: the pid that we are looking for
* @param shouldFreeNode: should we also free the memory allocated for this pcb
* struct
* @return true if successfully deleted, false if no match was found or error
*/
bool PCBSearchAndDelete(PCBDeque* deque, pid_t pgid, bool shouldFreeNode);

/**
* @brief Search a Deque for the first job with "STOPPED" status
*
* @param deque: the deque to search inside
* @return A pointer to an element in the deque that matches; NULL if no match
* found
*/
```

```
pcb* PCBDequeStopSearch(PCBDeque* deque);

/**
* @brief Search a Deque for the first job that is in the background
*
* @param deque: the deque to search inside
* @return A pointer to an element in the deque that matches; NULL if no match
* found
*/
pcb* PCBDequeBackgroundSearch(PCBDeque* deque);

extern PCBDeque* PCBList;  // make PCBList a global var
```

PIDDeque.h

```
#ifndef PIDDEQUE_H_
#define PIDDEQUE_H_

#include <stdbool.h>  // for bool type (true, false)
#include <sys/types.h>
#include "globals.h"

////////////////////////////////////////////////////////////////////////////////
// A Deque is a Double Ended Queue. We will implement a PID Deque which will
// be used to store the PIDs of the processes for our operating system. There
// will be a separate deque for each priority level in the system.
////////////////////////////////////////////////////////////////////////////////

/** @brief A single node within a deque.
*
* A node contains next and prev pointers as well as a pid.
*/
typedef struct pid_dq_node {
 pid_t pid;
 struct pid_dq_node* next;  // next node in deque, or NULL
 struct pid_dq_node* prev;  // prev node in deque, or NULL
} PIDDqNode;

// The entire Deque.
// This struct contains metadata about the deque.
typedef struct dq_struct {
 int num_elements;  //  # elements in the list
 PIDDqNode* front;  // beginning of deque, or NULL if empty
 PIDDqNode* back;   // end of deque, or NULL if empty
} PIDDeque;

// !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
// "Methods" for our Deque implementation.
// !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

/** @brief Allocates and returns a pointer to a new Deque.
*
* It is the Caller's responsibility to at some point call Deque_Free to free
* the associated memory.
*
```

```
* @return the newly-allocated deque, or NULL on error.
*/
PIDDeque* PIDDeque_Allocate(void);

/** @brief Free a Deque that was previously allocated by Deque_Allocate.
*
* @param deque the deque to free. It is unsafe to use "deque" after this
* function returns.
*/
void PIDDeque_Free(PIDDeque* deque);

/** @brief Return the number of elements in the deque.
*
* @param deque the deque to query.
* @return deque size.
*/
int PIDDeque_Size(PIDDeque* deque);

/** @brief Adds a new element to the front of the Deque.
*
* @param deque the Deque to push onto.
* @param payload the payload to push to the front
*/
void PIDDeque_Push_Front(PIDDeque* deque, pid_t payload);

/** @brief Pop an element from the front of the deque.
*
* @param deque the Deque to pop from.
* @param payload_ptr a return parameter; on success, the popped node's payload
* is returned through this parameter.
* @return false on failure (e.g., the deque is empty), true on success.
*/
bool PIDDeque_Pop_Front(PIDDeque* deque);

/** @brief Peeks at the element at the front of the deque.
*
* @param deque the Deque to peek.
* @param payload_ptr a return parameter; on success, the peeked node's payload
* is returned through this parameter.
* @return false on failure (e.g., the deque is empty), true on success.
*/
bool PIDDeque_Peek_Front(PIDDeque* deque, pid_t* payload_ptr);
```

```c
/** @brief Pushes a new element to the end of the deque.
*
* @param deque the Deque to push onto.
* @param payload the payload to push to the end
*/
void PIDDeque_Push_Back(PIDDeque* deque, pid_t payload);

/** @brief Pops an element from the end of the deque.
*
* @param deque the Deque to remove from
* @param payload_ptr a return parameter; on success, the popped node's payload
* is returned through this parameter.
* @return false on failure (e.g., the deque is empty), true on success.
*/
bool PIDDeque_Pop_Back(PIDDeque* deque);

/** @brief Peeks at the element at the back of the deque.
*
* @param deque the Deque to peek.
* @param payload_ptr a return parameter; on success, the peeked node's payload
* is returned through this parameter.
* @return false on failure (e.g., the deque is empty), true on success.
*/
bool PIDDeque_Peek_Back(PIDDeque* deque, pid_t* pid);

/** @brief Searches for a PID in the deque.
*
* @param deque the Deque to search within.
* @param pid the PID to search for.
* @return true if the PID is found, false otherwise.
*/
bool PIDDequeJobSearch(PIDDeque* deque, pid_t pid);

/** @brief Searches for a PID in the deque and deletes it if found.
*
* @param deque the Deque to modify.
* @param pid the PID to search and delete.
* @return true if the PID was successfully deleted, false otherwise.
*/
bool PIDSearchAndDelete(PIDDeque* deque, pid_t pid);
```

```
extern PIDDeque* priorityList[4];
#endif
```

terminal_history.h

```c
#ifndef TERMINAL_HISTORY_H
#define TERMINAL_HISTORY_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <unistd.h>

#define HISTORY_SIZE 500  // Maximum number of commands that can be stored

/**
* @brief Structure to hold the command history of a terminal session.
*
* This structure stores commands in a circular buffer, along with the indices
* for the current command, total number of commands, and the current position
* in the command history for navigation.
*/
typedef struct {
   char* commands[HISTORY_SIZE];  // Array of pointers to store the commands
   int current;                   // Index of the current command in the buffer
   int size;                      // Total number of commands stored
   int pos;                       // Current navigation position in the history
} TerminalHistory;

// Function prototypes

/**
* @brief Retrieves the next or previous command from the history based on navigation
input.
*
* @param history Pointer to the TerminalHistory structure.
* @param up Boolean flag to determine navigation direction; true for up (older
commands).
* @return Returns the command from the specified direction or NULL if no more
commands.
*/
char* get_history(TerminalHistory* history, bool up);
```

```
/**
* @brief Stores a command in the terminal history.
*
* @param history Pointer to the TerminalHistory structure.
* @param command String containing the command to store.
*/
void save_command(TerminalHistory* history, char* command);

/**
* @brief Reads the command history from a file.
*
* @return Returns a pointer to a newly allocated TerminalHistory structure with
history loaded from file.
*/
TerminalHistory* read_history_from_file();

/**
* @brief Frees the memory allocated for the terminal history and its commands.
*
* @param history Pointer to the TerminalHistory structure to be freed.
*/
void free_history(TerminalHistory* history);

#endif // TERMINAL_HISTORY_H
```