

Group 35 - Companion Document

Kernel Functions

Custom Struct

```
typedef struct p_info_struct {  
    int pid;  
    int status;  
    char * command;  
    int priority;  
    FD_LIST* fd_list;  
    int ground;  
} info;
```

void set_shell_pcb(pcb * p);

Parameters:

pcb * p - pcb pointer representing the shell PCB

Return:

Nothing

Functionality:

Sets the shell_pcb field to whatever p is

pcb * get_shell_pcb();

Parameters:

Nothing

Return:

Pointer to the shell's PCB

Functionality:

Getter for the shell's PCB

pcb * get_prev_pcb();

Parameters: None

Returns: a pointer to the *prev_pcb* variable

Functionality: gets the current *prev_pcb* from the kernel

pcb * get_prev_fg_pcb();

Parameters: None

Returns: a pointer to the *prev_fg_pcb* variable

Functionality: gets the current *prev_fg_pcb* from the kernel

void set_prev_pcb(pcb * new_prev);

Parameters: a pointer to a pcb

Returns: None

Functionality: sets the *prev_pcb* variable inside the kernel to the *new_prev* variable

void set_prev_fg_pcb(pcb * new_prev);

Parameters: a pointer to a pcb

Returns: None

Functionality: sets the *prev_ls_pcb* variable inside the kernel to the *new_prev* variable

pcb * get_neg();

Parameters:

Nothing

Return:

Pointer to the head of the -1 priority queue

Functionality:

Getter for the -1 priority queue

void set_neg(pcb * neg);

Parameters:

pcb * neg - pointer to the head of the -1 priority queue

Return:

Nothing

Functionality:

Setter for the -1 priority queue

pcb * get_zero();

Parameters:

Nothing

Return:

Pointer to the head of the 0 priority queue

Functionality:

Getter for the 0 priority queue

void set_zero(pcb * zero);

Parameters:

pcb * zero - pointer to the head of the 0 priority queue

Return:

Nothing

Functionality:

Setter for the - priority queue

pcb * get_one();

Parameters:

Nothing

Return:

Pointer to the head of the 1 priority queue

Functionality:

Getter for the 1 priority queue

void set_one(pcb * one);

Parameters:

pcb * one - pointer to the head of the 1 priority queue

Return:

Nothing

Functionality:

Setter for the 1 priority queue

pcb * get_zombies();

Parameters:

Nothing

Return:

Pointer to the head of the zombie queue

Functionality:

Getter for the zombie queue

pcb * get_cur_pcb();

Parameters:

Nothing

Return:

Pointer to the currently running PCB

Functionality:

Getter for the currently running PCB

void set_cur_pcb(pcb * new_pcb);

Parameters:

pcb * new_pcb - pointer to the new current PCB

Return:

Nothing

Functionality:

Setter for the current PCB

ucontext_t * get_scheduler_context();

Parameters:

Nothing

Return:

Pointer to the context of the scheduler

Functionality:

Getter for the scheduler context

void reset_scheduler();

Parameters:

Nothing

Return:

Nothing

Functionality:

Recreates scheduler context

void setup_scheduler();

Parameters:

Nothing

Return:

Nothing

Functionality:

Allocates memory for scheduler context

int get_timer_total();

Parameters:

Nothing

Return:

Number of clock ticks elapsed

Functionality:

Getter for the number of clock ticks elapsed

void incr_timer();

Parameters:

Nothing

Return:

Nothing

Functionality:

Increments the number of clock ticks

FILE * get_fptr();

Parameters:

Nothing

Return:

Pointer to the logfile

Functionality:

Getter for the logfile

void set_fptr(FILE * f);

Parameters:

FILE * f - pointer to the logfile

Return:

Nothing

Functionality:

Setter for the logfile file pointer

void fclose_fptr();

Parameters:

Nothing

Return:

Nothing

Functionality:

API to close the logfile upon successful exit

pcb* k_process_create(pcb* parent, ucontext_t * context);

Parameters:

pcb * parent - parent process

Ucontext_t * context - content for new child process

Return:

Pointer to the newly created PCB

Functionality:

Kernel component of creating new processes

void k_process_kill(pcb * process, int signal);

Parameters:

pcb * process - process to be updated by signal

int signal - signal used to update process

Return:

Nothing

Functionality:

Sends signal to process specified by PCB pointer

void k_process_terminate(pcb * process);

Parameters:

pcb * process - process to be terminated

Return:

Nothing

Functionality:

Called to terminate a process; sets status to zombie

pcb * k_get_current_process();

Parameters:

Nothing

Return:

Pointer to the currently running PCB

Functionality:

Kernel-side getter for the currently running PCB

void mkcontext(ucontext_t *uc, void *function, int argc, char const *argv[]);

Parameters:

ucontext_t * uc - empty context to be initialized

void * function - function which context will represent

int argc - number of function arguments

char const * argv[] - argument list for process

Return:

Nothing

Functionality:

Creates context for new processes, setting their uc_link to the scheduler

void k_process_swap_to_scheduler();

Parameters:

Nothing

Return:

Nothing

Functionality:

Forces a swap to the scheduler context

pcb * k_get_process_pid(int pid);

Parameters:

int pid - pid of the process to find

Return:

Pointer to the PCB with the specified pid

Functionality:

Finds the process with the specified pcb

void k_process_update_changed(pcb * process, int c);

Parameters:

pcb * process - process to be updated

int c - value to which "changed" flag is set

Return:

Nothing

Functionality:

Sets flag to indicate that process needs to be waited on

void k_process_nice(int p, int prior);

Parameters:

int p - pid of process to update

int prior - new priority for the process

Return:

Nothing

Functionality:

Updates priority of the process with the specified pid

info * create_info(int pid, int status, char * command, int priority, FD_LIST* fd_list, int ground);

Parameters:

int pid - pid of the process

int status - status of the process

char * command - command text of the process

int priority - priority of the process

FD_LIST * fd_list - file descriptor list of the process

int ground - ground type of the process

Return:

Custom struct containing process metadata

Functionality:

Constructor for custom struct to be returned by p_info

void free_info(info * info);

Parameters:

info * info - info struct to be freed

Return:

Nothing

Functionality:

Frees the info struct

info * k_info(int pid);

Parameters:

int pid - pid of the process about which you want info

Return:

info struct containing process metadata

Functionality:

Provides info about process without exposing PCB

info * k_get_current_info();

Parameters:

Nothing

Return:

info struct containing metadata about current process

Functionality:

Retrieves process metadata about currently running process

void k_end_everything();

Parameters:

Nothing

Return:

Nothing

Functionality:

Shuts down the operating system and provides cleanup

void setup_file();

Parameters:

Nothing

Return:

Nothing

Functionality:

Opens logfile pointer for writing

void setup_timer();

Parameters:

Nothing

Return:

Nothing

Functionality:

Sets up timer values, initialized to 100 ms for clock ticks

int get_timer_mod();

Parameters:

Nothing

Return:

int representing where you are in the current scheduler cycle

Functionality:

Used in determining which queue from which the scheduler should read processes

void incr_timer_mod();

Parameters:

Nothing

Return:

Nothing

Functionality:

Increments timer mod value

void mod_timer_mod();

Parameters:

Nothing

Return:

Nothing

Functionality:

Performs modulo on timer mod by 19 to maintain CPU balance

sigset_t * get_signal_mask();

Parameters:

Nothing

Return:

sigset_t pointer to the signal mask

Functionality:

Getter for the signal mask

void * get_signal_stack();

Parameters:

Nothing

Return:

void pointer to the stack of the idle context

Functionality:

Used in the initialization of the idle context

void set_signal_stack();

Parameters:

Nothing

Return:

Nothing

Functionality:

Allocates memory for the idle context's stack

void timeout();

Parameters:

Nothing

Return:

Nothing

Functionality:

Sets the flag for how the current process finished to TIMEOUT, which indicates that it was interrupted by the timer

void normal_finish();

Parameters:

Nothing

Return:

Nothing

Functionality:

Sets the flag for how the current process finished to NORMAL_FINISH, which indicates that it terminated within its time slice

int get_how_finished();

Parameters:

Nothing

Return:

int representing the how_finished flag

Functionality:

Getter to determine whether the current process terminated when the kernel swaps back to scheduler

void idle();

Parameters:

Nothing

Return:

Nothing

Functionality:

Suspends until it receives a SIGALRM signal from the timer interrupt

void setup_idle();

Parameters:

Nothing

Return:

Nothing

Functionality:

Sets up stack allocation for the idle context

void reset_idle();

Parameters:

Nothing

Return:

Nothing

Functionality:

Re-initializes the idle context

ucontext_t * get_idle();

Parameters:

Nothing

Return:

The idle context

Functionality:

Getter for the idle context

void scheduler();

Parameters:

Nothing

Return:

Nothing

Functionality:

Function containing the scheduler algorithm

pcb * get_queue(int queue);

Parameters:

int queue - 1, 0 or -1, depending on which queue you want to get

Return:

Pointer to the pcb of the specified respective queue

Functionality:

Gets the head of the priority queue specified by "queue"

void timer_interrupt();

Parameters:

Nothing

Return:

Nothing

Functionality:

Signal handler for SIGALRM; algorithm to handle when the timer interrupts

void setup_signals();

Parameters:

Nothing

Return:

Nothing

Functionality:

Sets up timer interrupt as the signal handler for the SIGALRM in order to handle timer interrupts

User Functions

Custom Struct

```
typedef struct p_wait_struct {  
    int pid;  
    int status;  
} p_wait_struct;
```

int W_WIFEXITED(int status);

Parameters:

int status - the status of the PCB in question

Return:

int - 1 (TRUE) if PCB exited naturally, 0 (FALSE) otherwise

Functionality:

Determines whether PCB status exited naturally

int W_WIFSTOPPED(int status);

Parameters:

int status - the status of the PCB in question

Return:

int - 1 (TRUE) if PCB is stopped, 0 (FALSE) otherwise

Functionality:

Determines whether PCB status is stopped

int W_WIFCONTINUED(int status);

Parameters:

int status - the status of the PCB in question

Return:

int - 1 (TRUE) if PCB is ready, 0 (FALSE) otherwise

Functionality:

Determines whether PCB status is ready

int W_WIFSIGNALED(int status);

Parameters:

int status - the status of the PCB in question

Return:

int - 1 (TRUE) if PCB was terminated by sigint, 0 (FALSE) otherwise

Functionality:

Determines whether PCB status was terminated by sigint

p_wait_struct * p_wait(int mode);

Parameters:

int mode: the kind of wait (hang, nohang)

Return:

p_wait_struct *: struct indicating the status and the pid of the process that was successfully waited on

Functionality:

Performs wait indiscriminately on every running process

HANG: If there is no immediate child to wait on, block the current process and context switch to scheduler. Else, it returns that child as the p_wait_struct

NOHANG: If there is no immediate child to wait on, return null. Else, it returns that child as the p_wait_struct

int p_spawn(void * func, int argc, char const *argv[], int ground);

Parameters:

void* func: function to spawn into a new thread

int argc: number of arguments in argv

char const *argv[]:

Int ground: foreground or background depending on ampersand

Return: the pid of the newly spawned process

Functionality: creates a new context based off the function and the arguments and then calls *k_process_create* in the kernel to create a new process.

void p_kill(int pid, int signal);

Parameters:

int pid: the pid of the process to kill

int signal: the signal used to modify the process in question

Return:

Nothing

Functionality:

Modifies the status of the pcb of the process with the pid "pid" with the signal "signal"

p_wait_struct * create_p_wait_struct(int pid, int status);

Parameters:

int pid: the pid of process to wait on

int status: the state of the child process

Return:

p_wait_struct*: a new struct with passed in parameters

Functionality:

Constructs a struct to be used as a return value for the wait function as per the specifications of the writeup. Holds the pid and the status

void p_exit();

Parameters: None

Return: None

Functionality: terminates the current process with a call to the kernel function *k_process_terminate*

void p_nice(int pid, int priority);

Parameters:

pid - given pid of process to change
priority - new queue to add process to

Return: None

Functionality: checks if priority is between 1 and -1. Calls *k_process_nice* to change the priority of given pid.

void empty();

Parameters:

Nothing

Return:

Nothing

Functionality:

Empty process spawned by *_sleep()*, just gets blocked

void _sleep(char * t_str);

Parameters:

char * t_str - number of seconds as passed to the sleep terminal command

Return:

Nothing

Functionality:

Spawns an empty process that just blocks for an arbitrary number of seconds

void p_sleep(int ticks);

Parameters:

int ticks - time (in seconds) process should sleep

Return:

Nothing

Functionality:

System call to allow a process to sleep

info * p_info(int pid);

Parameters: a pid

Returns: an Info struct corresponding to the pid

Functionality: find the PCB corresponding to the given pid and returns an info struct with a call to *create_info*

void zombie_child();

Parameters: None

Returns: None

Functionality: returns immediately

void busy();

Parameters:
None

Return:
None

Functionality:
Busy wait to eat CPU usage

void ps();

Parameters:
None

Return:
None

Functionality:
Prints all processes currently in the queue, including running processes, stopped processes and zombies

void shell_nice(char * priority, char * command, char * optional_arg);

Parameters:

priority - priority of new process
command - name of new function being spawned
optional_arg - arg for new function

Return:
None

Functionality:
Spawns a new function and then adds the new process to the given priority queue.

void zombify();

Parameters: None

Returns: None

Functionality: Used to test how our scheduler handles zombies. P_spawns zombie_child and then runs an infinite while loop.

void orphan_child();

Parameters: None

Returns: None

Functionality: runs an infinite while loop.

void orphanify();

Parameters: None

Returns: None

Functionality: spawns an orphan_child and returns right away. Used to test our scheduler.

void p_end_everything();

Parameters: None

Returns: Noen

Functionality: Called when control d is pressed. Makes a call to *k_end_everything* to end all of the processes and quit the shell.

```
int p_pid_present(int pid);
```

PCB Functions

Macros and Variables

TRUE 1
FALSE 0
RUNNING 0
BLOCKED 1
FINISHED 2
READY 3
ZOMBIE 4
ORPHAN 5
KILLED 6
STOPPED 7
READ 0
WRITE 1
BG 0
FG 1

Custom Struct

```
typedef struct pcb {  
    struct pcb * parent_pcb;  
    // parent_pcb is a pointer to the parent of a PCB. The parent of the shell points to null  
    int pgid;
```

```

// process group id
int ppid;
// parent's pid
int pid;
// pid of the process
ucontext_t * context;
// pointer to the context of the process
int priority;
// priority level of the process, can be -1, 0, or 1
char * command; // unsure if necessary
// read file descriptor
int bool_read;
// write file descriptor
int bool_write;
// status of process
int status;
// a flag to determine whether the PCB terminated naturally or due to a signal
int terminated_natty;
// file descriptor of PCB
int file_descriptor;
// pointer to the next PCB in the linked list
struct pcb * next;
// boolean to indicate whether a process has changed state; used in parent waiting
int child_changed;
// pointer to process' array of children processes
struct pcb ** child_list;
// number of children
int num_children;
// size of child array; used when we rescale the array
int child_list_size;
// flag to indicate if process status has changed. Used in p_wait
int changed;
// used in p_sleep. Value indicates how many seconds the process should sleep for
int sleep_time;
// per process file descriptor list
FD_LIST* fd_list
// indicates whether the process is running in the foreground or background
int ground;
// indicates whether the process was interrupted by any signal

```

```
    int signal_flag;  
} pcb;
```

Functions

pcb * create_PCB (int p, int pg, int pp, ucontext_t * con, int b_read, int b_write, int prior);

Parameters:
refer to the struct for the parameter descriptions

Return:
pcb just created

Functionality:
Creates a new PCB by instantiated all of the above fields

void set_ground(pcb * block, int new_ground);

Parameters: a pcb block and a ground field

Returns: nothing

Functionality: sets the ground field inside of block to new_ground.

pcb * find_next_valid(pcb * head);

Parameters:
the head of a list

Return:
the next valid pcb

Functionality:
We call this function when we want to determine the next process in a queue that is in the 'READY' state; if no processes are in the 'READY' state or the queue is empty, we return null.

pcb * add_PCB (pcb* head, pcb* newBlock);

Parameters:

head: the head of the list; newBlock: the pcb we want to add to the list

Return:

the head of the list, with newBlock inserted at the tail

Functionality:

Adds newBlock to the end of the list; if head is null, i.e. the list doesn't exist yet, we return newBlock as the head

pcb * add_to_head(pcb* old_head, pcb* new_head);

Parameters:

old head: we are replacing with new_head

Return:

the new head of the list

Functionality:

This method is only called in kill. It is used when a parent's child has terminated and now we are ready to change the parent's status -- we bring it to the front of the queue.

pcb * remove_PCB (pcb* head, pcb* remBlock);

Parameters:

head of list, the block we are trying to remove

Return:

the head of the new list without remBlock

Functionality: removes remBlock from the list and removes remBlock from its parent's child_list

pcb * find_PCB_pid(pcb* head, int p);

Parameters: head pcb, and a pid p that we are looking up

Return: the PCB in head with pid p or null if p doesn't exist

Functionality: iterates through the head list and returns the pcb if the corresponding pcb was found. Otherwise return null.

pcb * find_PCB_pgid(pcb* head, int p);

Parameters:

The head of the list, the pgid of the pcb we want to return

Return:

the pcb whose pgid matches p

Functionality:

returns the pcb whose pgid = p if it exists in the list. if it doesn't exist, returns null

pcb * find_PCB_ppid(pcb* head, int p);

Parameters:

The head of the list, the ppid of the pcb we want to return

Return:

the pcb whose ppid matches p

Functionality:

returns the pcb whose ppid = p if it exists in the list. if it doesn't exist, returns null

void add_child_PCB(pcb* block, pcb* child);

Parameters:

the parent (block) and its child

Return:

Nothing

Functionality:

Add child to block's child_list. We also increment the parent's num_children and possibly rescale the array if it is at capacity.

void free_PCB(pcb * block);

Parameters:

PCB we want to free

Return:

Nothing

Functionality:

Frees PCB from memory. Called in k_process_terminate().

void print_PCB(pcb * current);

Parameters: current pcb

Return:

Nothing

Functionality:

Prints the contents of current. Helpful for debugging.

void set_sleep(pcb * block, int sleep_time);

Parameters: a pcb block and an amount of time (sleep_time) the process will sleep for

Return: Nothin

Functionality: sets process status to blocked and sets the sleep_time field of block

void set_command(pcb * block, char * command);

Parameters: a block and a string command

Returns: nothing

Functionality: sets the command file in block to the string command

Job Functions

Custom Struct

```
struct Job {  
    int pid;  
    int current_job_number;  
    int bool_type; // for background or foreground  
    int last_modified_counter;  
    struct Job * next;  
    char * user_input;  
    int status; // tell whether its running or stopped  
    int num_processes;  
};
```

struct Job * create_job(int given_pid, int ground, int counter, char * input);

Parameters:

int given_pid - pid of the process which the job represents

int ground - determines whether job is foreground or background

int counter - int representing how recently the job was created

char * input - the user input which created the job

Returns:

struct Job * - a struct containing all of the process information pertinent for job control

Functionality:

Constructor for job control custom struct

void free_job(struct Job * j);

Parameters:

struct Job * j - job to be freed

Returns:

Nothing

Functionality:

Memory management for the job struct

void update_status(struct Job * job, int status, int counter);

Parameters:

struct Job * job - the job to update

int status - the new status of the job

int counter - indicates that job has been recently updated

Returns:

Nothing

Functionality:

Updates the status of the job in question

void update_ground_type(struct Job * job, int type, int counter);

Parameters:

struct Job * job - the job to update

int type - determines whether job should be in foreground or background

int counter - indicates that job has been recently updated

Returns:

Nothing

Functionality:

Updates ground type of the job struct

struct Job * add_job(struct Job * head, struct Job * new_job);

Parameters:

struct Job * head - the head of the jobs list

struct Job * new_job - the new job to add to the linked list

Returns:

The head of the job linked list

Functionality:

Adds a new job to the job list

struct Job * remove_job_index(struct Job * head, int num);

Parameters:

struct Job * head - the head of the jobs linked list

int num - the index of the linked list to remove

Returns:

The head of the job linked list

Functionality:

Removes the num-th job from the job list

void print_job(struct Job* job);

Parameters:

struct Job * job - job struct to be printed

Returns:

Nothing

Functionality:

Prints the job for debugging purposes

File System

Macros and Variables

Constants for f_open:

F_WRITE 1 // 001

F_READ 3 // 011

F_APPEND 7 // 111

Constants for f_lseek:

F_SEEK_SET 1 // 001

F_SEEK_CUR 3 // 011

F_SEEK_END 7 // 111

Constants for STD vals:

F_STDIN 0

F_STDOUT 1
F_STDERR 2

Relevant characters and strings:

TAB_CHAR "\t\t"
NEWLINE "\n"
INT_BUFF_LEN 20
LS_HEADER "name\t\tsize\t\tstart\n"
CAT_BUFFER_SIZE 4096

Permissions to open fs file

OPEN_MODE 0644

Return value on error

F_FAILED -1
F_SUCCESS 0

Constants for shell built-ins:

RC_MIN_SIZE 0
RC_MAX_SIZE 10000
RC_MIN_CHAR 33
RC_MAX_CHAR 125

Functions

void init_filesystem(const char* fs_name);

Parameters:

fs_name - the file path of the filesystem file

Return:

Nothing.

Functionality:

Initiates the global fields in filesystem.c that all other functions depend on. Also opens the filesystem file for reading and writing.

void free_filesystem();

Parameters:

Nothing.

Return:
Nothing

Functionality:
Frees all data structures related to the filesystem.

int f_open(const char * fname, int mode);

Parameters:

fname- file name

mode- F_WRITE, F_READ, F_APPEND

Return:
Integer. Used for determining a failure.

Functionality:
Open a file name fname with the mode mode and return a file descriptor on success and a negative value on error. Additionally, the file pointer references the end of the file.

int f_read(int fd, char * buf, unsigned int n);

Parameters:

buf - buffer for reading

n - size of buffer

Return:
Integer- Used for determining a failure.

Functionality:
Read n bytes from the file referenced by fd. On return, f_read returns the number of bytes read, 0 if EOF is reached, or a negative number on error.

void f_print_fds();

Parameters:

Nothing.

Return:

Nothing

Functionality:

Helper function to print the fds list.

int f_write(int fd, const char * str, unsigned int n);

Parameters:

fd - file descriptor

str - string to be written

n - size of string

Return:

Integer- Used for determining a failure.

Functionality:

Write n bytes of the string referenced by str to the file fd and increment the file pointer by n. On return, f_write returns the number of bytes written, or a negative value on error.

int f_close(int fd);

Parameters:

fd - file descriptor

Return:

Integer- Used for determining a failure.

Functionality:

Close the file fd and return 0 on success, or a negative value on failure.

int f_unlink(const char * fname);

Parameters:

fname - file name

Return:

Integer- Used for determining a failure.

Functionality:

Remove the file given the name.

int f_lseek(int fd, int offset, int whence);

Parameters:

fd - file descriptor.

offset - how much to offset by

whence - F_SEEK_SET, F_SEEK_CUR, F_SEEK_END

Return:

Nothing

Functionality:

Reposition the file pointer for fd to the offset relative to whence.

int f_dup2(int old_fd, int new_fd);

Parameters:

old_fd - old file descriptor

new_fd - new file

Return:

Nothing.

Functionality:

Dup2 implementation for redirections.

void f_dup_reset();

Parameters:

Nothing.

Return:

Nothing

Functionality:

Resets standard in/out/err of the current process to defaults.

void f_cat();

Parameters:

Nothing.

Return:

Nothing

Functionality:
Implementation of 'cat'.

void f_ls();

Parameters:
Nothing.

Return:
Nothing

Functionality:
Implementation of 'ls'.

void f_touch(char* fname);

Parameters:
fname - name of file to touch

Return:
Nothing

Functionality:
Implementation of 'touch'. If a file with name fname already exists, this
* function does nothing.

void f_rm(char* fname);

Parameters:
fname - name of file to remove

Return:
Nothing

Functionality:
Implementation of 'rm'. If no file with name fname exists, this function
does nothing. If the file with name fname has open file descriptors referring to it, it will
not be deleted until said descriptors are closed. (Will still delete when filesystem is
freed)

void f_reset();

***CUSTOM FUNCTION**

Parameters:

Nothing.

Return:

Nothing

Functionality:

Resets file system to new state.

void f_randchars(int size);

***CUSTOM FUNCTION**

Parameters:

size - expected range: [RC_MIN_SIZE, RC_MAX_SIZE]

Return:

Nothing

Functionality:

Writes 'size' random chars to S_STDOUT in the range [RC_MIN_CHAR, RC_MAX_CHAR].

void p_perror(const char* prefix);

Parameters:

prefix - prefix for perror message.

Return:

Nothing

Functionality:

Implementation of 'perror'. Uses the global constant in errors.c

Directory

Macros and Variables

```
DIR_SIZE_BYTES 4 // length of the directory file itself to start
NAME_BYTES     256
SIZE_BYTES     3
START_BYTES    1
FILE_ENC_BYTES (NAME_BYTES + SIZE_BYTES + START_BYTES)
NULL_TERM     '\0'
TRUE          1
FALSE         0
CHAR_MASK     0x000000FF
```

Custom Struct

```
typedef struct FILE_NODE {
    // the name of the file
    const char * name;
    // the size of the file
    unsigned int size;
    // the start block of the file in the fat table
    unsigned char start_block;
    // the number of open file descriptors referring to this file
    unsigned int open_fds;
    // a flag that, when set to true, means the file should be deleted when all file
    // descriptors that refer to it are closed
    int marked_for_deletion;
    // the next FILE_NODE in the FILE_LIST
    struct FILE_NODE * next;
} FILE_NODE;
```

Note: A FILE_NODE contains information about a file in the filesystem, as well as a pointer to the next file in the filesystem.

```
typedef struct FILE_LIST {
    // the first FILE_NODE in the linked list
    FILE_NODE *head;
    // the size of the directory file encoding (not including the first four bytes, which stores
    // this value in the directory file)
    unsigned int dir_size;
} FILE_LIST;
```

Note: A FILE_LIST contains a pointer to the head of a FILE_NODE linked list, containing all the files in the filesystem (not including the directory file).

Functions

FILE_LIST* init_file_list();

Parameters:

Nothing

Return:

FILE_LIST that was initialized.

Functionality:

Initializes a file list. Not to be confused with load_file_list, which loads a file list from the filesystem file. This function is just a helper function for that one.

FILE_NODE* init_file_node(const char* name, int size, unsigned char start_block);

Parameters:

name - Name for file node

size - the size of the file referenced by this FILE_NODE

start_block - the start block of the file referenced by this FILE_NODE

Return: FILE_NODE that was initialized

Functionality:

Initializes a file node with given values.

FILE_LIST* load_file_list(int fs_fd);

Parameters:

fs_fd - file system file descriptor

Return:

FILE_LIST that was loaded.

Functionality:

Parses the directory file and loads all the file information into a FILE_LIST

unsigned int get_encoding_size(int fs_fd);

Parameters:

fs_fd - file system file descriptor

Return:

Unsigned int - the size of the encoding

Functionality:

Get the size of the directory encoding

unsigned int get_directory_file_size(int fs_fd);

Parameters:

fw_fd - file system file descriptor

Return:

unsigned int - the size of the directory file

Functionality:

Reads the first DIR_SIZE_BYTES of the first file in the filesystem and interprets them as an integer denoting the length of the directory file.

char* get_directory_encoding(int fs_fd, unsigned int directory_size);

Parameters:

fs_fd - file system file descriptor

directory_size - size of directory

Return:

char * - the encoding that was stored in the directory file

Functionality:

Read the entire directory file starting AFTER the first DIR_SIZE_BYTES bytes.

FILE_LIST* load_list_from_encoding(char* list_encoding, unsigned int encoding_size);

Parameters:

list_encoding - the encoding of the file list (returned by get directory encoding)

encoding_size - the size of the encoding (a multiple of FILE_ENC_BYTES)

Return:

FILE_LIST that was loaded.

Functionality:

Loads the file list information encoded in list_encoding into a FILE_LIST.

FILE_NODE* load_file_node_from_encoding(unsigned char* node_encoding);

Parameters:

node_encoding - the encoding of one file node

Return:

FILE_NODE* loaded.

Functionality:

Loads the file information encoded in node_encoding into a FILE_NODE. Expected encoding: "[name][size][start_block]"

FILE_NODE* find_node_with_name(FILE_LIST* list, const char* name);

Parameters:

list - given file list

name = name of node to find

Return:

FILE_NODE * found

Functionality:

Looks through the list to find a file node with a matching name. If match is found, return NULL.

void add_file(int fs_fd, FILE_LIST* list, FILE_NODE* node);

Parameters:

fs_fd - file system descriptor

list - given file system list

node - file node to be added

Return:

Nothing.

Functionality:

Add FILE_NODE* node to the end of the given file list. This function will also update the FAT and append the new file encoding to the directory file.

unsigned int export_file_list(int fs_fd, FILE_LIST* list);

Parameters:

fs_fd - file system file descriptor

list - list to be exported

Return:

unsigned int - number of bytes written

Functionality:

Encodes the file list into the directory file. Returns the new length of the list encoding.

unsigned int export_file(int fs_fd, FILE_NODE* node, unsigned int* cursor);

Parameters:

fs_fd - file system file descriptor

node - given file node

cursor - a pointer to the current offset into the directory file -- when exporting the whole list, export_file starts writing where that last export_file left off

Return:

unsigned int - number of bytes written

Functionality:

Encode the file node and appends it to the end of the directory file.

int read_from_file(int fs_fd, unsigned char start_block, unsigned int file_size, unsigned int* cursor, char* buf, unsigned int read_size);

Parameters:

fs_fd - filesystem file descriptor

start_block - the start block of the file to read from

file_size - the size of the file

cursor - pointer to the cursor offset

buf - where to store the newly-read bytes

read_size - the number of bytes to read

Return:

int - the number of bytes read

Functionality:

Read from the file that starts at start_block. This function will update the value referenced by cursor to its new spot in the file

void remove_file(FILE_LIST* file_list, FILE_NODE* file, int fs_fd);

Parameters:

file_list - given file list

fs_fd - file system file descriptor

Return:

Nothing

Functionality:

Removes the file from the file system and updates the FAT and directory.

**int write_to_file(int fs_fd, unsigned char start_block, unsigned int file_size,
 unsigned int* cursor, const unsigned char* buf, unsigned int write_size);**

Parameters:

fs_fd - filesystem file descriptor

start_block - the start block of the file to read from

file_size - the size of the file

cursor - pointer to the cursor offset

buf - where to store the newly-written bytes

write_size - the number of bytes to write

Return:

int - the number of bytes writtens

Functionality:

Write to a file starting at cursor. This function does not have a reference to the FILE_NODE, the caller must update the file_size themselves: file_size = max{cursor, file_size}

/**

* @brief Frees the given file list.

*

* @param the list to free

*/

void free_file_list(FILE_LIST* list);

Parameters:

list - given list

Return:

Nothing

Functionality:

Frees the given file list.

void free_file_node(FILE_NODE* node);

Parameters:

node the node to free

Return:

something

Functionality:

something;

int mark_file_for_deletion(FILE_NODE* file, FILE_LIST* file_list, int fs_fd);

Parameters:

file - given file to be marked

file_list - the FILE_LIST to remove the node from if the node is deleted

fs_fd - file system file descriptor

Return:

Integer - TRUE(1) if the node was removed, FALSE(0) otherwise

Functionality:

Marks a file for deletion. If there are no open descriptors referencing this file, the file is removed.

void update_dir_size(int fs_fd, FILE_LIST* list, unsigned int new_dir_size);

Parameters:

list - given list

fs_fd - file system file descriptor

dir_size - new directory size

Return:

Nothing

Functionality:

Sets the `dir_size` field in the file list to `new_dir_size`. Also writes the new dir size to the front of the dir file.

unsigned int shift_chars_into_int(unsigned char* buf, unsigned int n);

Parameters:

char - the string of chars to shift into the int

N- the number of chars in buf

Return:

unsigned int - the int represented by the chars

Functionality:

Shift chars into int. Used for decoding directory file

void truncate_file(int fs_fd, FILE_LIST* list, FILE_NODE* file);

Parameters:

fs_fd - file system file descriptor

list - given list

file - given file node

Return:

Nothing

Functionality:

Sets the file size to zero and free any blocks it used to take up in the FAT.

Descriptors

Macros and Variables

SUCCESS 0

FAILURE -1

FD_LIST_SIZE 64 The max number of allowed file descriptors per process

MIN_FREE_FD Smallest available fd number (0,1,2 are taken)

Custom Struct

```
typedef struct FD_NODE {  
    // the name of the file this file descriptor refers to  
    char* fname;  
    // F_READ | F_WRITE | F_APPEND  
    int mode;  
    // the number of fds in the FD_LIST that refer to this FD_NODE (dup)  
    int open_fds;  
    // the current offset into the file  
    unsigned int cursor;  
} FD_NODE;
```

Note: This represents all the information corresponding to the file descriptor fd. This includes the mode (e.g. F_READ | F_WRITE) and the cursor.

```
typedef struct FD_LIST {  
    // the global list of FD_NODE*s  
    FD_NODE** fds;  
} FD_LIST;
```

FD_LIST* init_fd_list(FILE_LIST* file_list);

Parameters:

File_list - given file list

Return:

FD_LIST* that was initialized

Functionality:

Initializes a new FD_LIST and returns a pointer to it.

void free_fd_list(FD_LIST* list);

Parameters:

list - given list

Return:

Nothing

Functionality:

Frees the given FD_LIST. This will not free the FILE_NODE referenced by list -> file

void free_fd_node(FD_NODE* node);

Parameters:

node - given node

Return:

Nothing

Functionality:

Frees the given FD_NODE

int get_new_fd(FILE_LIST* file_list, FD_LIST* fd_list, FILE_NODE* file, int mode);

Parameters:

file_list - the file_list to look through for the FILE_NODE to update it

fd_list -

file - the file for the FD_NODE to refer to

mode - F_READ | F_WRITE | F_APPEND

Return:

The new file descriptor

Functionality:

Get a file descriptor for the specified file.

FD_NODE* create_fd_node(FILE_LIST* file_list, char* fname, int mode);

Parameters:

fname - the name of the file that this file descriptor refers to

mode - F_READ | F_WRITE | F_APPEND

Return:

The new FD_NODE

Functionality:

Create a new FD_NODE

void set_fd_node(int fd, FD_LIST* list, FD_NODE* node);

Parameters:

fd - the index in the FD_LIST to set

list - the FD_LIST

node - the FD_NODE for (list -> fds)[fd] to refer to

Return:

Nothing

Functionality:

Inserts the node into the list, sorted by fd

int remove_fd_node(int fd, FD_LIST* list);

Parameters:

fd - file descriptor

list - given list

Return:

Int -

Functionality:

Remove the FD_NODE with the specified fd. This frees the FD_NODE and sets the pointer to NULL. If the actual FD_NODE is removed (read freed), return TRUE.

int find_free_fd(FD_LIST* list);

Parameters:

list - given list

Return:

int -

Functionality:

Looks through the list and returns the lowest file descriptor available \geq list->min

FD_NODE* get_node_by_fd(int fd, FD_LIST* list);

Parameters:

fd - file descriptor

list - given list

Return:

Node found

Functionality:

Find a FD_NODE with the given fd. If no match is found, return NULL.

FD_NODE* create_dummy_fd_node(FILE_LIST* file_list);

Parameters:

file_list - give list

Return:

something

Functionality:

Create a dummy FD_NODE with null file pointer and mode = 0

Fat

Macros and Variables

FAT_SIZE 256

BLOCK_SIZE 1024

FREE_BLOCK 0

DIR_BLOCK 0

NULL_BYTE 0

Functions

void write_to_terminal(char* str);

Parameters:

str

Return:

Nothing

Functionality:

Printf but for FS team. Just debugger method.

unsigned char get_next_block(int fs_fd, unsigned char index);

Parameters:

fs_fd - filesystem file descriptor

index - index of block to get

Return:

unsigned char

Functionality:

Get the block at the specified index

void set_next_block(int fs_fd, unsigned char index, unsigned char value);

Parameters:

fs_fd - filesystem file descriptor

index - index to set new value

value - new value to be set

Return:

Nothing

Functionality:

Set the value at the specified index

void lseek_to_block(int fs_fd, unsigned char block, unsigned int offset);

Parameters:

fs_fd - filesystem file descriptor

block - given block

offset [0, BLOCK_SIZE) -- offset within the block

Return:

Nothing

Functionality:

lseek to the offset of the block in the fs file

unsigned char find_free_block(int fs_fd);

Parameters:

fs_fd - filesystem file descriptor

Return:
unsigned char -

Functionality:
Returns the first block after 0 that maps to a 0.
* If no such block is found, 0 is returned.

void free_blocks(int fs_fd, unsigned char start_block);

Parameters:
fs_fd - filesystem file descriptor
start_block -

Return:
Nothing.

Functionality:
Starting at start_block, follow all next-block pointers, setting blocks to FREE_BLOCK in your wake, until a block points to itself.

Shell

Custom Struct

```
typedef struct history_struct {  
    char * cmd;  
    struct history_struct * next;  
} hist_node;
```

hist_node * create_hist_node(char * cmd);

Parameters:
char * cmd - user input to STDIN

Return:
A wrapper struct containing cmd and a pointer to another hist_node struct

Functionality:
We call this function to create a new hist_node pointer, which we use as a part of a linked list that contains all of the user's input to STDIN. The contents of this linked list are printed on a call to history()

hist_node * add_hist(hist_node * head, hist_node * new_node);

Parameters:

hist_node * head - the head of the history linked list

hist_node * new_node - the new node to add to the linked list

Return:

The head of the linked list with the new node added to the tail

Functionality:

Records the latest input to STDIN in a linked list of nodes, each of which contain one instance of STDIN input

void signal_handler(int signo);

Parameters:

int signo - integer representing the signal sent by the user

Return:

Nothing

Functionality:

Signal handler for ^Z and ^C

int has_ampersand(char* user_input);

Parameters:

char * user_input - the user's input to STDIN

Return:

int - 1 (TRUE) or 0 (FALSE)

Functionality:

Returns TRUE if user_input contains an ampersand and FALSE if it does not

char check_input(char * user_input);

Parameters:

char * user_input - the user's input to STDIN

Return:

char - null character if valid input, the character which causes user_input to fail the validity check if it is not

Functionality:

Checks to see if user_input to STDIN is a valid input, i.e. contains no pipes, contains the correct number of redirects (if any), contains the correct number of ampersands (if any)

void man();

Parameters:

None

Return:

None

Functionality:

Prints a list of every executable command

void history();

***CUSTOM FUNCTION**

Parameters:

None

Return:

None

Functionality:

Prints a complete record of all user input into STDIN for that session

void touch(char* filename);

Parameters:

char * filename - name of the file to be created

Return:

None

Functionality:

Creates a new file with name "filename"

void rm(char* filename);

Parameters:

char * filename - name of the file to be removed

Return:

None

Functionality:

Removes the file with name "filename"

int shell();

Parameters:

None

Return:

int - exit code (0 if successful, 1 otherwise)

Functionality:

Code for the interactive shell