PennOS Documentation - Group 19

Raunaq Abhyankar, Swachhand Lokhande, Jeremy Saslaw, Abhishek Tiwari

File System

```
/**
  * @brief opens file in file_system
  *
  * @param fname: The file's name to open
  * @param mode: The mode to open file in (-r for read, -w for write, -a for append)
  * @return int: file's file descriptor upon success, -1 upon failure
  */
int f_open(char *fname, int mode)
```

f_open function allows for the user to open a file in a given mode for subsequent read/write/append/seek calls to it.

```
/**
  * @brief Reads from a file in the file system and stores up to n bytes in buf
  *
  * @param process_fd: file descriptor of the given process to read from
  * @param n: maximum number of bytes to read
  * @param buf: the buffer to store the bytes in
  * @return int: number of bytes read and stored in buf
  */
int f_read(int process_fd, int n, char *buf)
```

Normal read function, that enables reading from a file.

```
/**
  * @brief writes from a string into a given file up to n bytes
  *
  * @param process_fd: file descriptor of the given process to write to
  * @param str: the string to write from
  * @param n: maximum number of bytes to read
  * @return int: number of bytes written to file
  */
int f_write(int process_fd, const char *str, int n)
```

f_write function will take in a string and an amount of bytes n, and then writes to the given process_fd. This can be used to write to STDOUT and STDERR as per the typical write system call. Errors will occur when trying to write from STDIN.

```
/**
  * @brief closes a given file
  *
  * @param process_fd: file descriptor to close
  * @return int: 1 upon success, -1 upon failure
  */
```
int f_close(int process_fd)

f_close function will close a file from being used when it is open. If the file is already closed, f_close will do nothing.

```
/**
  * @brief unlinks (deletes) a given file
  *
  * @param fname: file to unlink
  * @return int: 1 upon success, -1 upon failure
  */
```
int f_unlink(char *fname)

f_unlink unlinks (deletes) a file from the file system. If the file is open by any process at the time, f_unlink will notify the user and not allow for the file to be deleted. Be careful, as this is a permanent process, when unlinking a file, it is unrecoverable in the future.

```
/**
 * @brief seeks within a file
 *
 * @param process_fd: file descriptor of process to seek in
 * @param offset: offset how far in file to seek
 * @param whence: where in file to seek from (F_SEEK_SET, F_SEEK_CUR, F_SEEK_END)
 * @return int: 1 upon success, -1 upon failure
 */
```
int f_lseek(int process_fd, int offset, int whence)

Usage : works like the lseek function, used to seek to the desired location in a file.

```
/**
  * @brief prints to desired file descriptor using printf argument types
  *
  * @param output_fd: where to print to (can be STDOUT_FILENO or STDERR_FILENO too)
  * @param format_string: string with formatting allowed
  * @param ...: arguments that can be used with format_string, similar to printf
  * @return int: 1 upon success, -1 upon failure
  */
```
int f_printf(int output_fd, char *format_string, ...)

f_printf is PennOS's substitute for printf. The first argument allows you to select where to write to (including STDOUT and STDERR). The second argument is a format string, where typical format codes like %d and %s are allowed. Finally, the rest of the arguments substitute in for the format codes written in format_string.

```
/**
  * @brief prints error message
  *
  * @param header_error_string: pre-string to the error message
  * @param x: error code
  */
```

void f_error(char *header_error_string, int x)

f_error is PennOS's substitute for perror. The first argument allows the user to type a string (or "" if none is desired), and the second allows for the user to select a predefined error code. These include:

```
#define FILENOTFOUND 0
#define CANTCLOSEOPENFILE 1
#define CANTOPENFILESYSTEM 2
#define CANTLSEEK 3
#define CANTREADFILENODE 4
#define ERRORWRITING 5
#define FSFULL 6
#define BADREADSTDOUT 7
#define INVALIDCOMMAND 8
#define NOPROCESS 9
#define PROCESSRUNNING 10
```

Kernel

```
/**
* @brief Will fork a new thread that retains most of the attributes
* of the parent thread (using k_process_create). Once the thread is spawned,
* it will execute the function referenced by func with the command arguments
* provided as an array of strings in the command_args_t object cmd
* (see command_args_t).
*
* @param func The function to execute
* @param cmd The command arguments to be given to the function
*
* @return int pid of the new process
*/
int p_spawn(void* func, command_args_t* cmd);

/*
// Struct representing redirection features.
// isValid: determine if the redirection command is valid
// in_file, out_file: standard in_file and out_file for redirection command
*/
typedef struct command_args_t {
   enum validationMsg isValid;
   char *in_file, *out_file;
   int mode;
   char *cmd;
   char **argv;
   int argc;
   int cmdTokens;
   int is_bg;                  // is the process being run in bg
   int priority;
} command_args_t;
```

```
/**
* @brief Kill the thread referenced by pid with the signal 'signal'
*
* @param pid The pid of the process to kill
* @param signal_no The signal to send  (S_SIGTERM, S_SIGSTOP or S_SIGCONT)
*
* @return int SUCCESS if the kill was successful, FAILURE otherwise
*/
int p_kill(int pid, int signal_no);



/**
* @brief Waits for a child process to change it's state.
*
* Returns immediately if called with mode == NOHANG, blocks otherwise
*
* @param mode NOHANG or !NOHANG
* @return wait_res a struct containing the pid and the reason p_wait
* returned. If there are no children to wait on it sets wait_res.pid to -1
*/
wait_res p_wait(int mode);


typedef struct wait_res {
   int pid;
   int status;
} wait_res;
pid: PID of the returned process
status: Reason why p_wait returned


/**
* @brief Exits the current process unconditionally
*
* @return int SUCCESS or FAILURE
*/
int p_exit();
```

```
/**
* @brief Return true if the child terminated normally
*
* @param status wait_res.status
* @return int
*/
int W_WIFEXITED(int status);


/**
* @brief return true if the child was stopped by a signal.
*
* @param status wait_res.status
* @return int
*/
int W_WIFSTOPPED(int status);


/**
* @brief return true if the child was continued by a signal.
*
* @param status wait_res.status
* @return int
*/
int W_WIFCONTINUED(int status);


/**
* @brief return true if the child was terminated by a signal, that is,
*  by a call to p_kill with the S_SIGTERM signal.
*
* @param status wait_res.status
* @return int
*/
int W_WIFSIGNALED(int status);
```

```
/*
* @brief Set priority level of process pid to priority
* @param pid Integer value of pid
* @param priority value to be assigned
*
* @return void
*/
```
void p_nice(int pid, int priority);

Set the priority of process specified by ‘pid’ as ‘priority

```
/*
* @brief Return pcb_t with all relevant information of
* process pid
* @param pid Integer value of pid
*
* @return pcb_t* Struct with all the relevant information
*/
```
struct pcb_t* p_info(int pid);

Return a structure representing standard information about a thread pid. We simply return the process’ pcb_struct

```
/*
* @brief Put thread to sleep for ticks
* @param pid Integer value of pid
* @param priority value to be assigned
*
* @return void
*/
```
void p_sleep(int ticks);

Set the process specified by ‘pid’ to Blocked until ‘ticks’ of the clock occur.

```
/*
* @brief Creates a new pcb struct for the new process.
* New process inherits open file descriptors from parent. Makes context for
* new process, make entry in global process table and adds it to scheduler
* queue for scheduling.
*
* @param parent Pointer to parent's pcb
* @param func The function to execute
* @param cmd Pointer to parsed command_args_struct
*
* @return int pid of the new process
*
*/
```

int k_process_create(struct pcb_t *parent, void *func, command_args_t* cmd);

Creates a new process, child of parent with specified function and parsed input arguments cmd.

```
/*
* @brief Kill the specified process with the signal 'signal'
*
* @param process Pointer to process to kill
* @param signal_no The signal to send (see kernel.h)
*
* @return int SUCCESS if the kill was successful, FAILURE otherwise
*/
```

int k_process_kill(struct pcb_t *process, int signal_no);

Send ‘signal_no’ to ‘process’

```
/*
* @brief Send S_SIGSTSP to specified process
*
* @param process Pointer to process to kill
*
* @void
*/
```

void k_send_sigstop(struct pcb_t *process);

Send SIGSTSP to ‘process’

```
/*
* @brief Send S_SIGCONT to specified process
*
* @param process Pointer to process to kill
*
* @void
*/
```
void k_send_sigcont(struct pcb_t *process);

Send SIGCONT to ‘process’

```
/*
* @brief Send S_SIGTERM to specified process
*
* @param process Pointer to process to kill
*
* @void
*/
```
void k_send_sigterm(struct pcb_t *process);

Send SIGTERM to ‘process’

```
/*
* @brief Terminate a process. Change its state to ZOMBIE, delete it
* from parent's child_q and insert into parent's zombie_q. Delete its children
* recursively
*
* @param process Pointer to process to terminate
*
* @int SUCCESS or FAILURE
*/
```
k_process_terminate(pcb_t * process)

Kernel design:

The kernel has several data structures.

process_table: Global process table that contains the pid, pcb, pointer to the nodes in the scheduler queues. We support a maximum of 10000 concurrent processes. But this can be changed easily in kernel.h

3 scheduler queues for each priority levels implemented as circular linked lists

sleep_q: A queue that contains all the sleeping processes. At each tick, we decrement the ticks in the pcb of all the processes in this queue

curr_proc: This keeps track of the current running process

fg_prog: This keeps track of the foreground process to implement terminal control

bg_q: This queue keeps track of the background processes

The pcb contains the following fields:

```
ucontext_t *uc;                     // user context info
ucontext_t *term_uc;                // context for termination
pid_t pid;                          // process id
struct pcb_t *parent_pcb;           // parent process pcb pointer
int priority;                       // priority level of the process
int status;                         // current status of the process
int prev_status;                    // the previous status of the process (used for
wait)
int ticks;                          // number of clock cycles to sleep
pcb_queue *zombie_q;                // queue for zombied children
pcb_queue *child_q;                 // queue of child processes
int normal_term;                    // did the process terminate normally?
int waiting_on_child;               // is the process waiting on the child?
int is_bg;                          // is the process being run in bg?
command_args_t *cmd;                // the command this process is running
open_file_table_t *file_info;        // File table information for given process
```

To implement wait, we keep track of the previous state of the process. When wait is called without NOHANG, we block the process and set waiting_on_child to true. If a child changes state, it unblocks the parent if the parent's waiting_on_child is true. The parent does the necessary cleanup and returns the pid and the reason of status change

Shell

```
/**
* @brief Used in the shell for lsFlatFAT
*
* @param args : arguments for ls
*/
```
void ls(void* args)
Usage: lists all files in the current directory

```
/**
  * @brief creates new file
  *
  * @param args: Arguments to the function
  */
```
void touch(void* args)
touch simply takes a file input, and creates a new file in the file system. If the file system is full, an error is returned.

```
/**
  * @brief removes file
  *
  * @param args: Arguments to the function
  */
```
void rm(void* args)
rm removes a file from the file system. If the file doesn't exist, nothing is done.

```
/**
  * @brief reads or writes based on syntax
  *
  * @param args: Arguments to the function
  */
```
void cat(void* args)
cat performs similarly to the terminal's cat, with reading and writing functionalities. If the file doesn't exist, cat creates it, and allows the user to write to it before an EOF. If cat is called alone with no arguments, then the user can write and print to the terminal. Redirections allow for useful cat functionality to and from functions.

```
/**
```

```
 * @brief counts words, new_lines, and bytes of file or STDIN
 *
 * @param args: Arguments to the function
 */
```
void wc(void* args)

Usage: If a input file is specified, else takes input from the STDIN, and counts the words in the text. Can write to a file if specified else it writes to the STDOUT.

```
/**
  * @brief lists available commands for shell
  *
  * @param args: Arguments to the function
  */
```
void man(void* args)

man is a very simple command, which allows for the user to see all the possible commands of the shell, along with their necessary arguments.

```
/**
  * @brief reverses a string
  *
  * @param args: Arguments to the function
  */
```
void reverse(void* args)

reverse takes a string as an argument, and reverses it for the user. This can be redirected to a file for useful possible applications.

```
/**
* @brief Used for sort in the shell
*
* @param args : arguments for sort
*/
```
void my_sort(void* args)

Usage: If a input file is specified, else takes input from the STDIN, and sorts the words in the text lexicographically. Can write to a file if specified else it writes to the STDOUT.

```
/**
* @brief Shell builtin for fg
*
* @param args
*/
```
void fg(void *args)

Usage: Brings a process to foreground.

```
/**
* @brief Shell builtin for bg
*
* @param args : arguments for bg
*/
```

void bg(void *args)

Usage: Moves the process to background.

```
/*
 * @brief Change priority/niceness of process specified by
 * pid. Involves moving nodes between scheduler queues.
 *
 * @param args : arguments for nice_pid
 */
```

my_nice_pid(void *args)

Adjust the nice level of process with ‘pid’ to specified ‘priority’.

```
/**
* @brief Shell builtin for exiting
*
*/
```

void my_logout()

Usage: Use to logout of the system

```
/**
* @brief Function for sleep. Spawn a process.
*
* @param args : arguments for the spawned process
*/
```

void my_sleep(void* args)

Usage: creates a process that sleeps for the specified time.

```
/**
* @brief Shell builtin for displaying all current jobs
* Running, blocked and stopped jobs
*
* @param args
```

```
*/
```
void jobs(void *args)

Print all running, blocked and stopped jobs in the system.

```
/**
* @brief Print information of current processes in the system.
*
* @param args : arguments for the spawned process
*/
```
void ps(void* args);

Print information of current processes in the system

```
/*
* @brief Spawn a command with specified priority value
*
* @param void
*/
```
void exec_with_nice(void* args);

Spawn a command with specified priority and execute it