Penn OS Code Documentation

CIS 5480 Spring 2023
Group 55: Qi Xue, Ze Sheng, Zhiqi Cui, Zhengjia Mao

Foreword

PennOS is a UNIX-like operating system that runs as a guest OS within a single process on a host OS. PennOS models standard UNIX, including kernel features, FAT file system, and a user shell. This file serves as a companion document to PennOS and PennFAT, and provides necessary information to understand the project architecture, as well as the declared data structures, variables, and functions. This document also specifies the purposes of each function and explains the passed-in arguments. More specifically, the project is structured as below.

src: all the source files
- **pennos**: all the source files for the kernel and shell, and a standalone executable pennOS that interacts with the file system and kernel via shell.
- **pennfat**: all the source files for the file system, and a standalone executable pennFAT that interacts with the file system.

bin: all the binary files to execute, which are ./pennos and ./pennfat.
log: the txt files that log all command history
doc: companion documents, including this one.

Catalog

1. pennos
    a. process_control.c
    b. process_control.h
    c. process_list.c
    d. process_list.h
    e. scheduler.c
    f. scheduler.h
    g. ucontext_func.c
    h. ucontext_func.h
    i. job_handler.c
    j. job.h
    k. logger.c
    l. logger.h
    m. parser.h
    n. shell-behavior.c
    o. shell-behavior.h
    p. shell.c
    q. shell.h
    r. pennos.c

    s. stress.c
    t. stress.h
2. pennfat
    a. fat.c
    b. fat.h
    c. file.c
    d. file.h
    e. pennfat_handler.c
    f. pennfat_handler.h
    g. utils.c
    h. utils.h
    i. pennfat.c
3. macros.h

pennos

process_control.c

This file contains the process-like functions that operate on PCBs in both kernel mode and user mode.

- **static void alarm_handler(int signum)**
  This is used for handling the SIGALARM. In this function, we increase the cpu_timer and we iterate the whole sleeping list and update sleep special list for process remaining sleep time:
  If the sleeping process has remaining time less than 0 and the process status is "BLOCKED", then we remove it from the blocked queue and sleep list, and add it into the ready queue with lower priority, and also we update the status as "READY". And when the time quantum finishes, we should swap to the scheduler context.

- **static void k_set_alarm_handler(void (*handler)())**
  When we handle sigalarm, suspend all other signals.
  void (*handler)(): the passed-in handler, which is alarm_handler().

- **static void k_set_timer(void)**
  Setup the clock

- **static void k_free_stacks(void)**
  Free up the memory

- **int k_process_control_initiate()**
  Initialize the scheduler, which is a processlist type defined in process_list.h, initialize the ucontext and kernel threads, set up the alarm handler, p_spawn the shell as the current process.

- **int k_process_control_start()**
  Set up the clock and jump to the scheduler.

- **PCB* k_process_create(PCB* parent)**
  Create a new PCB with its ucontext, allocate the first (minimum) available pid, create a new ucontext and inherit the stdin and stdout and open fds from the parent.
  PCB* parent: the parent PCB to inherit information from

- **int k_process_cleanup(PCB* process)**
  Destroy a pcb and its ucontext.
  PCB* process: the process PCB to cleanup.

- **int k_process_kill(PCB* process, int signal)**
  If receive S_SIGCONT, move the stopped process from stopped queues to ready queues, and set it to READY.
  If receive S_SIGSTP, stop the process, set it to STOPPED and move it to the stopped queue, and unblockly wait for it (put it in the background).
  If receive S_SIGTERM, terminate the process
  PCB* process: the process PCB to perform kill on
  int signal: the signal used to perform kill

- **void k_unblock_wait(processlists* pl, PCB* pcb)**
  Get the pid of the given pcb, find one in waiting list waiting for this exited process (given pcb).
  processlists* pl: the process list
  PCB* pcb: the PCB

- **int k_terminate(PCB* pcb, bool signaled)**
  This function is called by p_exit(), p_kill(), and the special returner thread that handles the normal return of any process.
  This function is called under two situation (indicated by bool signaled):
  1. (signaled=false) terminates normally: by returner or p_exit();
  2. (signaled=true) terminates by signal: by k_process_kill(), this can happen when we recursively handle the childrens of one existing process.

  This function follows:
  (The PCB is either deleted inside waitpid (waiting zombies), or inside this function if we identify it as an Orphan)
  - if the process is already terminated, do nothing and return;
  Then check signaled flag:
  - signaled
    If is not signaled (return normally):
      - move pcb to zombie list
      - change the pcb status to EXITED
      - check waiting list to unblock process
      - scan in PL, find all children, update status to ORPHAN in their pcb, and k_process_kill(SIGTERM) them (and k_process_kill would call k_terminate) - it will be the recursion process - it is 1.singaled=true and 2.status=ORPHAN
      - return
    If is signaled: (caused by SIGTERM)
    - check pcb status:
      - if is ORPHAN: (zombies are also marked as ORPHAN)
        - destroy PCB and clean up p_nodes; (no need to check waitpid, because parents die)
        - scan in PL, find all children, update status to Orphan and k_process_kill(SIGTERM) them (the same: k_process_kill will call k_terminate inside)
        - return
      - else if is READY or BLOCKED:
        - move pcb to zombie list
        - change the pcb status to SIGNALED
        - check waiting list to unblock process
        - scan in PL, find all children,update status to ORPHAN, and k_process_kill(SIGTERM) them (k_process_kill will call k_terminate inside) - and it is 1.singaled=true and 2.status=ORPHAN
        - return

- **void kernel_thread_scheduler()**
  We first check if the ready queue is empty or not.
  If it is not, we call k_deterministic_shcedule() to schedule a process to run, and then we delete the process in the ready_queue and move it to the tail.
  If it is empty, then we run idle, which means the shell is blocking waiting for someone, thus, we just change to the idle context.

- **void kernel_thread_returner()**
  Help to manually call k_terminate() on just returned process

- **void kernel_thread_idle()**
  Help to wait.

- **pid_t p_spawn(void (*func)(), char* argv[], int fd0, int fd1)**
  The wrapping function of k_process_create(), accessible by user land.
  It creates a new pcb for the shell or the command, make ucontext for the newly spawned process, and put the new process to the assigned ready queue. It will be put to the medium priority ready queue by default.
  void (*func)(): the function for the ucontext
  char* argv[]: the argument for the ucontext
  int fd0: fd in
  int fd1: fd out

- **void p_sleep(unsigned int ticks)**
  This function moves the current process to the blocked queue and adds it to the sleeping list.
  unsigned int ticks: the number of seconds to sleep

- **pid_t p_waitpid(pid_t pid, int* wstatus, bool nohang)**
  The function handles the process with the given pid.
  1. Check validity. Invalid if the given pid isn't in the zombie queue or the current process has no child process.
  2. Find the waited pid node in the zombie queue / the stopped queue.
  3. If we don't find the waited pid node,
  If it is in (nohang=true) non-block mode, it will directly return.
  If it is in (nohang=false) block mode, it will block the current process until we find the waited pid node.
  4. If we find the waited pid node, update the process status.
  Return:
    - pid if waitpid successfully returned and found a stopped or zombied child
    - 0 if waitpid is in non-blocking version, and no status change can found
    - -1 If wait pid fails, e.g. invalid pid, no-child, or unexpected error happen

  pid_t pid: the process id to wait for
  int* wstatus: the status of the process
  bool nohang: whether it is blocking

- **int p_kill(pid_t pid, int sig)**
  The wrapping function of k_process_kill, accessible by user land.
  pid_t pid: the process if to perform kill
  int sig: the signal, or the type of kill

- **int p_nice(pid_t pid, int priority)**
  1. Check whether the pid is valid.
  2. Change the priority: if the process is in the ready queue, this function will move it to the new ready queue based on the new priority.
  pid_t pid: the process id to change the priority
  int priority: the priority to set

- **void p_exit(void)**
  Exit the current process unconditionally.

- **bool W_WIFEXITED(int wstatus)**
  Check whether the process exits normally.
  int wstatus: process status

- **bool W_WIFSTOPPED(int wstatus)**
  Check whether the process is stopped.
  int wstatus: process status

- **bool W_WIFSIGNALED(int wstatus)**
  Check whether the process is terminated due to signals.
  int wstatus: process status

- **void sig_handler(int signo)**
  The relay function that translates system signals SIGINT, SIGSTP to our own S_SIGINT, SIGSTP.
  If receive SIGINT,
  If the current pcb is shell_pcb, we ignore it and create a new line.
  Else, we call p_kill and send S_SIGTERM to the foreground one.

  If receive SIGSTP:
  If the current pcb is shell_pcb, we ignore it and create a new line.
  Else, we call p_kill and send S_SIGSTP to the foreground one.
  int signo: the system signal to relay

- **void k_sig_reg()**
  Setup the signal handler for SIGINT and SIGTSTP signal: For both signals, they only will be sent to the process which has terminal control. 1. For SIGINT signals caused by Ctrl-C, it will terminate the currently running process except for the shell. 2. For SIGTSTP signals caused by Ctrl-D, it will stop the currently running process except for the shell.

- **void k_enter_protected_mode()**
  When the system call is running, it should not be interrupted by the SIGALRM or other signals. The reason for this is that inside system calls, some kernel procedures may manipulate the OS internal data (for example, the linked list of Ready Queues), and stop at some random point during such procedures may break the data in the kernel. Therefore, we set up a flag to differentiate the kernel mode(cannot be interrupted) and the user mode(can be interrupted). To make the protection to work recursively, we use a stack to realize the nested protection. When the OS enters the protected mode, we set the sigmask to block the signals, and the old sigmask is also pushed into the protection stack. When it leaves the protected mode, it goes back to the previous level mode

- **void k_leave_last_protected_mode()**
  When it leaves current mode, it gets the last mode info from the stack. This function returns to the mode stored in the stack.

- **void p_logout()**
  Exit the shell. Used when the user enters Ctrl-D or “exit” or “logout”.

- **PCB *get_current_PCB()**
  Helper function to return the current process.

- **processlists *get_scheduler()**
  Helper function to return the scheduler, which is a processlist.

process_control.h

This organizes ucontext threads and other crucial information into PCB and treats them like processes. It also packs PCB into pNode that is used to construct linkedlist-like queues, then it packs the queues into processLists.

- **struct PCB**
  The data structure that wraps ucontext and stores all other information needed to run the process-like thread.
- **struct pNode**
  The data structure that wraps a PCB as a node, along with next and prev, used to construct a doubly-linked list.
- **struct processLists**
  The data structure that wraps multiple linked lists, used by scheduler.
- **enum STATE**
  EXITED = 0,
  STOPPED = 1,
  BLOCKED = 2,
  READY = 3,
  SIGNALED = 4,
  ORPHAN = 5,
  RUNNING = 6,
  WAITING = 7,
  ZOMBIED = 8
- **enum Priority**
  HIGH = -1,
  MED = 0,
  LOW = 1,

process_list.c

This file contains the linkedlist-like functions that operate on pNode and processList in both kernel mode and user mode.

- **pnode* k_create_list_node(PCB* pcb)**
  Create a process node for the given PCB.
  PCB* pcb: the PCB to be wrapped as a node.

- **pnode* k_add_head(pnode* head, PCB* pcb)**
  Add the process node to the head of the queue.
  pnode* head: the queue head
  PCB* pcb: the node to add

- **pnode* k_add_tail(pnode* head, PCB* pcb)**
  Add the process node to the tail of the queue.
  pnode* head: the queue head
  PCB* pcb: the node to add

- **pnode* k_delete(pnode* head, PCB* pcb)**
  Delete the process in the given queue.
  pnode* head: the queue head
  PCB* pcb: the node to delete

- **pnode* k_get_node_by_pid_in_single_list(pnode* head, pid_t pid)**
  Find the process in the specified queue given the pid.
  pnode* head: the queue head
  pid_t pid: the node's pcb id to get

- **pnode* k_get_node_by_ppid_in_single_list(pnode* head, pid_t pid)**
  Find the child process in the specified queue given the parent pid.
  pnode* head: the queue head
  pid_t pid: the node's pcb's parent id

- **processlists* k_create_process_lists()**
  This creates and initializes the process lists we maintained and the 8 queues in the process pool.

- **pnode* k_get_node_by_pid_in_multiple_lists(processlists* pl, pid_t pid)**
  Find the process in the process lists given pid of the process, we would call k_get_node_by_pid_in_single_list to search in every queue in process lists
  processlists* pl: the processlists
  pid_t pid: the node's pcb id to get

- **pid_t k_find_available_pid(processlists* pl)**
  This function is called when we spawns a new process. We should find an available pid which is not used by another process.
  processlists* pl: the processlists

- **void k_delete_in_processlists(processlists* pl, pid_t pid)**
  Delete the process in the process lists.
  processlists* pl: the processlists
  pid_t pid: the node's pcb id to delete

process_list.h

This file only declares the functions from process_list.c

scheduler.c

This file runs the scheduler on the processlists.

- **int k_schedule(process_pool_t* plists)**
  Dynamically maintain a recent scheduled priority record by a fixed length queue, count in this queue, greedily select the priority queue according to the ratio compared with the standard ratio to make the gap decrease.
  process_pool_t* plists: the processlists

scheduler.h

This file only declares functions from scheduler.c

ucontext_func.c

This file provides our own helper functions that make context and set stack.

- **static void k_set_stack(stack_t* stack)**
  Register the stack.
  stack_t* stack: the stack to register and setup.

- **static void k_make_context(ucontext_t* ucp, ucontext_t* next, void (*func)(), char** argv, int mode)**
  1. Get the context.
  2. Set the signal mask.
  If the ucontext is for kernel, it should block most signals (not blocking SIGALARM).
  If the ucontext is for user, it should receive all signals.
  3. Link the next context. (for user apps next ucontext to execute always be returner
)
  4. Make the context.
  ucontext_t* ucp: the ucontext to make
  ucontext_t* next: the next ucontext to link
  void (*func)(): the function to make context
  char** argv: the argument to make context
  int mode: the mode (kernel/user)

ucontext_func.h

This file only declares the functions from ucontext_func.c.

job_handler.c

This file performs all operations related to job controls.

- **void job_printer(job_node *job, int opt)**
  This function prints out all the jobs in the background
  job_node* job: the head of the job node linked list
  int opt: the option

- **job_node* add_job(job_node *head, int *pids, int num_commands, struct parsed_command* cmds)**
  Create a new job based on the passed-in commands and adds it to the job queue and return the tail.
  job_node* head: the head of the job node linked list
  int *pids: the pids of the job node
  int num_commands: the number of commands
  struct parsed_command *cmds: the data structure that contains all information related to the commands

- **void remove_job(job_node *job)**
  Delete the given job from the job queue.
  job_node* job: the job node to delete

- **void free_job(job_node *deleted_job);**
  Free up the memory used by a deleted job..
  job_node* deleted_job: the job node to free

- **job_node *initialize_queue();**
  Initialize a job queue by creating a new head and return the head.

- **job_node *find_job_by_pid(job_node *head, int pid);**
  Find the job node in the queue by pid.
  job_node* head: the head of the job node linked list
  int pid: the pid of the job node

- **job_node *change_job_status(job_node *changed_job, int status, int changed_pid);**
  Update the status of a job node given its pid.
  job_node* head: the head of the job node linked list
  int pid: the pid of the job node
  int status: the status to set

- **void check_if_finished(job_node *head)**
  Check the finished jobs
  job_node* head: the head of the job node linked list

- **void restart_job(job_node *job)**
  Change a stopped job to running when "fg" is called.
  job_node* job: the stopped job node to continue

- **void display_job(job_node *head)**
  Print out the jobs when "jobs" is called
  job_node* head: the head of the job node linked list

- **job_node *bg(job_node *head, char **cmd_res)**
  Resume a stopped background job. If a job id is not given, find the current job or the most recently stopped one. Otherwise, find the job by job id and resume it.
  job_node* head: the head of the job node linked list
  char **cmd_res: the command arguments to find the correct node

- **job_node *fg(job_node *head, char **cmd_res)**

Restart and bring a background job to the foreground. If we don't give the job id, operate the most recently stopped node, otherwise the most recent one. If stopped, restart it and bring it to foreground. If running, bring it to the foreground and wait for the foreground jobs to complete.
job_node* head: the head of the job node linked list
char **cmd_res: the command arguments to find the correct node

job.h

This file declares the variables and functions from job_handler.c, and a job data sturcture.

- **struct job_node**
  The data structure that stores all information about a job, in a linked list format.
  struct job_node *prev;
  struct job_node *next;
  int job_id;
  int job_group_id;
  int job_num;
  int *job_pids;
  char *job_status;
  char *job_status_list;
  struct parsed_command *job_cmds;
  int fd_in;
  int fd_out;

shell-behavior.c

This file defines a list of built-in commands that our shell supports. The commands below are used with p_spawn, and some interact with the mounted file system, and some interact with the kernel.

- **void cmd_cat(char **argv);**
  Concatenate and display the contents of files.

- **void cmd_sleep(char **argv);**
  Sleep for the given seconds.

- **void cmd_busy(char **argv);**
  Hang the shell (keep it busy) and use the CPU.

- **void cmd_echo(char **argv);**
  Print a message to the terminal.

- **void cmd_ls(char **argv);**
  List the contents of a directory.

- **void cmd_touch(char **argv);**
  Create a new empty file or update the modification time of an existing file.

- **void cmd_mv(char **argv);**

Move or rename a file or directory.

- **void cmd_cp(char **argv);**
  Copy a file or directory.

- **void cmd_rm(char **argv);**
  Remove a file or directory.

- **void cmd_chmod(char **argv);**
  Change the permissions of a file or directory.

- **void cmd_ps(char **argv);**
  Show a list of running processes.

- **void cmd_kill(char *argv[]);**
  Terminate a process with customized signals.

- **void zombie_child();**
  The function used for **zombify()**

- **void zombify();**
  Make the current process a zombie process.

- **void orphan_child();**
  The function used for **orphanify()**

- **void orphanify();**
  Make the current process an orphan process.

- **void cmd_man();**
  Display the manual page for a command

- **void cmd_cd(char *args[]);**
  Change the current working directory.

- **void cmd_mkdir(char *args[]);**
  Create a new directory.

shell-behavior.h

This file only declares the functions from shell-behavior.c

shell.c

This file contains a shell function that the user can interact with. It takes the user input, parses the commands, supports the commands: hang, nohang, recur, test, exit, logout, fg, bg, jobs, man, nice_pid, nice, mkfs, mount, unmount, cat, sleep, busy, echo, ls, touch, mv, cp, rm, chmod, ps, kill, zombify, orphanify.

- **void shell()**
  The shell function that can be used in p_spawn().

shell.h

This file only declares the functions from shell.c

parser.h

This file is a provided parser that parses user arguments into commands and related information.

- **struct command_t**
  This struct stores all necessary information about user inputs in order to support multiple commands, pipeline, and redirections.
  bool is_background;
  bool is_file_append;
  const char *stdin_file;
  const char *stdout_file;
  size_t num_commands;
  char **commands[];

pennos.c

This is the main file that outputs an executable binary file to run. It is based on the kernel, shell, and file system.

- **int main(int argc, char *argv[])**
  Initialize the scheduler, wake up the shell.

Pennfat

To develop an operating system named PennOS, each process will be transformed into a context thread and ucontext library will be used for thread creation, destruction, and scheduling. Additionally, a process control block (PCB) structure needs to be designed and implemented to store various information about running threads such as process ID, parent process ID, children process IDs, open file descriptors, priority, etc. PCB structure needs to be carefully considered for various scenarios to ensure the correctness and reliability of the operating system. Finally, detailed documentation of the design and implementation process is required, including the description of the PCB structure and the design and implementation details of relevant algorithms.

Fat.c

initDirEntryNode(char *fileName, uint32_t size, uint16_t firstBlock, uint8_t type, uint8_t perm, time_t time):
Initializes a directory entry node and returns it.

freeDirEntryNode(dirEntryNode *fNode):
Frees the memory of a directory entry node.

initFat(char *fileName, uint8_t totalBlocks, uint8_t blockSizeIndex):
Initializes a FAT table and returns it.

ovewirteFat(char *fileName, uint8_t totalBlocks, uint8_t blockSizeIndex):
Overwrites an existing FAT table.

loadDirEntries(pennfat *fat):
Loads all directory entries in a specified FAT table.

loadFat(char *fileName):
Loads a specified FAT table from a file and returns it.

saveFat(pennfat *fat):
Saves a specified FAT table to the disk.

freeFat(pennfat **fat):
Frees the memory of a specified FAT table.

Fat.h:

This file contains structs for dirEntry and pennFat and all functions in fat.c

File.c

void freeFile(file *file):
Frees the memory allocated for a file.

void getDirEntryNode(dirEntryNode **prev, dirEntryNode **target, char *fileName, pennfat *fat):
Gets the directory entry node of a specified file.

file *getAllFile(pennfat *fat):
Gets all files in a file system.

uint8_t *getContents(uint16_t startIndex, uint32_t len, pennfat *fat):
Gets the contents of a file at a specified starting index and length.

file *readFile(char *fileName, pennfat *fat):
Reads the contents of a specified file.

void deleteFileHelper(dirEntryNode *prev, dirEntryNode *entryNode, pennfat *fat, bool dirFile):
Helper function for deleting a file.

int deleteFile(char *fileName, pennfat *fat, bool flag):
Deletes a specified file.

int renameFile(char *oldFileName, char *newFileName, pennfat *fat):
Renames a specified file.

int writeFile(char *fileName, uint8_t *bytes, uint32_t offset, uint32_t len, uint8_t type, uint8_t perm, pennfat *fat, bool flag, bool syscall, bool writeDir):
Writes bytes to a specified file.

int appendFile(char *fileName, uint8_t *bytes, uint32_t len, pennfat *fat, bool flag):
Appends bytes to a specified file.

int writeDirEntries(pennfat *fat):
Writes the directory entries to a file system.

int chmodFile(pennfat *fat, char *fileName, int newPerms):
Changes the permissions of a specified file.

int f_open(const char *filename, int mode):
Opens a file with a specified mode and returns a file descriptor on success or a negative value on error.

int f_close(int fd):
Closes a file with a specified file descriptor and returns 0 on success or a negative value on failure.

void f_ls(const char *filename):
Lists all files in the current directory or the directory of a specified file.

int f_read(int fd, int n, char *buf):
Reads n bytes from a file with a specified file descriptor and returns the number of bytes read, 0 if EOF is reached, or a negative number on error.

int f_write(int fd, const char *str, int n):
Writes n bytes from a specified string to a file with a specified file descriptor and returns the number of bytes written or a negative value on error.

int f_unlink(const char *filename):

Deletes a specified file and returns 0 on success or a negative value on failure.

int f_lseek(int fd, int offset, int whence):
Changes the position in a file with a specified file descriptor and returns 0 on success or a negative value on failure.

int f_mv(char *src, char *dest):
Moves a file from a source location to a destination location and returns 0 on success or a negative value on failure.

int f_cp(char *src, char *dest):
Copies a file from a source location to a destination location and returns 0 on success or a negative value on failure.

int f_chmod(char *filename, int newPerms):
Changes the permissions of a specified

File.h

This file contains all declaration of functions in file.c

Pennfat_handler.c

int pennfatMkfs(char *fileName, uint8_t numBlocks, uint8_t blockSizeIndex, pennfat **fat)
Initializes a new FAT file system on a file with the given name, number of blocks, and block size index, and returns a pointer to the newly created file system.

int pennfatMount(char *fileName, pennfat **fat)
Mounts an existing FAT file system stored in a file with the given name, and returns a pointer to the mounted file system.

int pennfatUnmount(pennfat **fat)
Unmounts the currently mounted FAT file system and saves any changes made to it.

int pennfatTouch(char **files)
Creates empty regular files with the given names in the currently mounted FAT file system.

int pennfatMove(char *oldFileName, char *newFileName)
Renames a file with the given old name to the given new name in the currently mounted FAT file system.

int pennfatRemove(char **files)
Deletes files with the given names from the currently mounted FAT file system.

int pennfatCat(char **commands, int count)
Concatenates the contents of the files with the given names, and either prints the result to stdout or writes it to a file.

int pennfatCopy(char **commands, int count, bool copyingFromHost, bool copyingToHost)
Copies a file to or from the currently mounted FAT file system, either from or to the host file system, depending on the values of the copyingFromHost and copyingToHost parameters.

int pennfatLs()
Lists the files and directories in the currently mounted FAT file system.

int pennfatChmod(char **commands, int perm)
Changes the permissions of a file with the given name in the currently mounted FAT file system to the given value.

Pennfat_handler.h

This file contains all declaration of functions in file.c

pennfat.c

The standalone pennfat shell that calls on the file system commands.

Utils.c, Utils.h

void writeHelper(char *message)
A helper function for printing something.

Other Files

macros.h

This file contains all macros used across the project, for the purpose of consistency.