PennOS

Project 2 Group 11

https://github.com/CIS548/22fa-project-2-group-11
Reifon Chiu, Sumukh Govindaraju, Jio Jeong, Rowena Lu

File Structure

```
bin/
	pennfat
	pennos
log/
	scheduler.log
doc/
	CompanionDoc.pdf
src/
	fat/
		fat_util.c
		fat_util.h
		fat.c
		file_kernel_funcs.c
		file_kernel_funcs.h
	kernel/
		os.c
		pcb_table.c
		pcb_table.h
		process_kernel_funcs.c
		process_kernel_funcs.h
		scheduler.c
		scheduler.h
		threads.c
		threads.h
	lib/
		directory_entry.c
		directory_entry.h
		errno.c
		errno.h
		fd.c
		fd.h
		file_system.h
		linked_list.c
```

```
            linked_list.h
            log.c
            log.h
            macros.h
            parser.h
            pcb.c
            pcb.h
            signals.h
            status.c
            status.h
      shell/
            commands.c
            commands.h
            job.c
            job.h
            redirects.c
            redirects.h
            shell.c
            shell.h
      user/
            file_user_funcs.c
            file_user_funcs.h
            process_user_funcs.c
            process_user_funcs.h
            scheduler_user_funcs.c
            scheduler_user_funcs.h
            stress.c
            stress.h
.gitignore
Makefile
parser-aarch64.o
parser-x86_64.o
README.md
```

System Calls

Process Related System Calls

`pid_t p_spawn(void (*func) (), char *argv[], int fd0, int fd1)`

This function calls k_process_create and returns the `pid` of the created `child_process`. If the `child_process` is `NULL`, then the return value is `-1`. Additionally, `fd0` and `fd1` are

the file descriptors of the input and output file respectively. This can be found in `process_user_funcs.c` .

`pid_t p_waitpid(pid_t pid, int *wstatus, bool nohang)`

If the given pid is -1, wait on any child that changes status. Otherwise, wait on the child with the given pid. If there isn't a child process for the parent to wait on, then the return value is `-1`. If `nohang` is set to `true`, and if any child or the specified child has changed status, return the pid. Otherwise, return 0. If `nohang` is set to `false`, then the current process is blocked until the child changes state, at which point the `pid` is eventually returned. This function also has various logging events depending on the child's `status` and whether `nohang` is `false`. This can be found in `process_user_funcs.c` .

`int p_kill(pid_t pid, int sig)`

This function calls k_process_kill, and sends the given signal to the process with the given pid. Returns `0` upon success. If the process with the given `pid` is not found, return `-1`. This can be found in `process_user_funcs.c` .

`void p_exit(void)`

This calls k_process_kill with a S_SIGTERM and exits the current thread unconditionally. If the active job is null, the function returns sets the error number and returns. This can be found in `process_user_funcs.c` .

`int p_nice(pid_t pid, int priority)`

This sets the priority of the thread `pid` to `priority`. This returns `0` if there are no errors. This returns -1 when the given priority is not -1, 0, or 1 and when there is no process with the given pid in the process table. This can be found in scheduler`_user_funcs.c` .

`void p_sleep(unsigned int ticks)`

This sets the calling process to be blocked for `ticks` ticks, and then sets the thread to running. If the active job is null, the function returns sets the error number and returns. This can be found in scheduler`_user_funcs.c` .

File System System Calls

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

This opens a file with the name `fname` in the mode `mode`, and returns a file descriptor of that file. The mode can be any one of three modes: `F_WRITE` or `0`, `F_READ` or `1`, and `F_APPEND` or `2`. For `F_WRITE`, the file is opened for reading and writing and is overwritten, and returns an error if the same file is being opened more than once. For `F_READ`, the file is opened only for reading, and returns an error if the file does not exist. For `F_APPEND`, the file is opened for reading and writing but does not overwrite the existing data. Also note that for `F_APPEND` the pointer is at the end of the file. Additionally, `F_WRITE` and `F_APPEND` creates the file if it does not exist. Lastly, this returns a file descriptor upon

success, but a negative value in the case that there's an error (invalid characters in filenames, file is not able to be created, etc.).

`int f_read(int fd, int n, char *buf)`
This returns the number of `n` bytes referenced by the file descriptor `fd`. If `EOF` is reached, this returns `0`. If there's an error with reading the file (ex: file is not opened / not found), this returns `-1`.

`int f_write(int fd, const char *str, int n)`
This returns the number of `n` bytes referenced by the string `str` that is written to the file descriptor `fd`, and increments the file pointer by `n`. This returns the number of bytes written, or `-1` in the case of an error (ex:file is not opened / not found, or there is no more space).

`int f_close(int fd)`
This closes the file `fd` and returns `0` upon success, but `-1` upon failure.

`int f_unlink(const char *fname)`
This removes the file `fname` and returns `0` upon success, but `-1` if the file has not been closed.

`int f_lseek(int fd, int offset, int whence)`
This sets the file pointer for `fd` to be the `offset` relative to `whence`. Additionally, `whence` can be any of the constants `F_SEEK_SET`, `F_SEEK_CUR`, and `F_SEEK_END`, which are the beginning of the file, the current position of the file pointer, and the end of file respectively. This returns `0` upon success, and `-1` in the case of an error (ex: file is not opened / not found).

`int f_ls(const char *filename)`
This lists the file `filename` in the directory, but lists all the files in the current directory if `filename` is `NULL`. This returns `1` if the filename is found, otherwise it returns `0`.

PCB Description

The PCB contains several fields:

Current status of the job.
`process_status status`

Whether the status has changed or not since last calling waitpid() on this process.
`bool status_changed`

`Whether the process is blocked or not`

bool blocked

`Whether the process is in the background or not`
bool is_bg

Context of the process.
ucontext_t* context

PID for the process.
pid_t pid

Parent process's PCB.
struct pcb_st* parent

Dynamically allocated string representing the pipeline name.
char* cmd

Linked list of zombie child processes.
linked_list* zombieLL

LInked list of nonzombie child processes.
linked_list* childLL

Priority level.
int priority

Array of open file descriptors with fd[0] = STDIN and fd[1] = STDOUT.
int fd[1024]

`Dynamically allocated and null terminated array of string arguments passed to the process.`
char argv**

Note also that the PCB's process_status can be any of the following: `STOPPED`, `RUNNING`, `ZOMBIED`, `TERMINATED`, `EXITED`, and `ORPHANED (please see status.h for more details)`.

Process Table Description

The process table is a linked list that contains pcb data structures of all processes (see `pcb_table.c`)

The processes associated with the process table is the following:

`void init_pcb_table()`
This function initializes the linked list for the process table.

`void add_pcb_to_table(pcb* to_add)`
This function adds the specified pcb into the process table.

`pcb* find_pcb_in_table(int pid)`
This function finds the pcb of the process specified by the input pid within the process table.

`void delete_pcb_in_table(int pid)`
This function deletes the pcb of the process specified by the input pid within the process table.

`linked_list* get_process_table()`
This function returns the address of the process table.

`void free_process_table()`
This function frees all the dynamically allocated variables (pcb and fields of pcb) associated with the process table.

PCB Related Functions

In `pcb.c` , there are the following functions:

`bool pid_equal_predicate(void* pid, void *target_pcb);`
This is used as a `predicate` in a `linked_list` function that searches for the `linked_list_elem` with the given `pid` in the given `pcb`.

`pcb* k_process_create(pcb* parent)`
This creates the child thread along with its `PCB`. Additionally, it retains the properties of the `parent`, and returns a reference to the new `PCB`. In the case of a `malloc` error, `NULL` is returned and an error message is printed. This can be found in `process_kernel_funcs.c` .

`void k_process_kill(pcb *process, int signal)`
This sends the signal `signal` to the `process`, and logs the event. If the signal is S_SIGTERM, it zombifies the target process and it orphans all of its children by iterating through its children linked lists and setting their status as ORPHANED). This can be found in `process_kernel_funcs.c` .

`void k_process_cleanup(pcb* process)`

This function iterates through the initial `process` 's children and if they're in the zombie linked list or in the non-zombie linked list, the children are removed from the lists and their `PCB` 's are `free`d. In the case that the children's `status` are neither `TERMINATED`, `EXITED`, nor `ORPHANED`, an error is printed stating that the process is not a zombie. This function helps to clean the memory that's allocated from `k_process_create()`. This returns nothing. This can be found in `process_kernel_funcs.c` .

`void free_process_pcb(pcb *process)`

This function goes through a process pcb frees each of its values as long as they aren't NULL.

`bool process_complete(pcb* j)`

This function returns whether the `pid` of the pcb `j` is equal to `FINISHED_PID_VAL`.

The remaining functions have to deal with the linked list which is used for the job queues.

`pcb* get_job_from_pid(linked_list *linked_list, int pid)`

This function returns the process `pcb` given a `pid`.

`job* get_current_job(linked_list *linked_list)`

This function returns the last stopped background job, or the last background job if nothing is stopped. The background job queue is passed in as a linked list.

`void update_job_status(linked_list *linked_list, int pid, int status)`

This function changes the `status` of a job to the given `status` , given the `pid` of the job.

Signal Description

The signals we used were the default signals as described in the writeup. These signals are: `S_SIGSTOP`, `S_SIGCONT`, and `S_SIGTERM`.

Error Numbers

```
    // Process/kernel errors
    NOERROR, // No error
    NOCHILDCREATED, // k_process_create failed to create a shell
    ACTIVEJOBNULL, // Passed in job is null
    NOTINPCBTABLE, // Job is invalid and not in PCB Table.
    KILLZOMBIE, // Tried to send signal to non-existent/zombie process.
    NOSUCHCHILD, // Parent doesn't have child with PID.
    NOTFOUNDINSCHEDULER, // Job could not be found in the scheduler with the
right priority
    INVALIDPRIORITY, // Invalid priority integer
    INVALIDSIGNAL, // Invalid signal integer passed to p_kill

    // File kernel errors
    INVALID_WHENCE, // Invalid whence value for f_lseek
    INVALID_OFFSET, // Invalid offset for f_lseek
    FILE_NOT_FOUND, // File is not found in file user function
    UNALLOCATED_BLOCK, // File does not have an allocated first block
    PERMISSION_DENIED, // User does not have permissions to access the file

    // File user errors
    INVALID_MODE, // Invalid integer file mode
    INVALID_FILE_NAME, // File name is NULL
    INVALID_FILE_NAME_POSIX, // File name does not follow POSIX standards
    ATTEMPTED_DOUBLE_WRITE, // Trying to open again in write mode
    READ_FILE_NOT_FOUND, // f_open File not found and was opened in F_READ
mode
    CLOSE_UNOPEN_FILE, // f_close File is not open
    DOUBLE_DELETION, // File has already been deleted
    FILE_NOT_FOUND_OFT, // fd not found in OFT

    // Critical Errors
    NO_MORE_SPACE // no more space in the fs
```

Description of Remaining Files

fat/
    fat_util.c, fat_util.h, fat.c, file_kernel_funcs.c, file_kernel_funcs.h
kernel/
    os.c, pcb_table.c, pcb_table.h, process_kernel_funcs.c, process_kernel_funcs.h, scheduler.c, scheduler.h, threads.c, threads.h
lib/
    directory_entry.c, directory_entry.h, errno.c, errno.h, fd.c, fd.h, file_system.h, log.c, log.h, macros.h, parser.h, signals.h, status.c, status.h
shell/
    commands.c, commands.h, job.c, job.h, redirects.c, redirects.h, shell.c, shell.h
user/
    scheduler_user_funcs.c, scheduler_user_funcs.h, stress.c, stress.h

Description of Other Data Structures

Directory Entry

This is a struct that we created that contains data pertaining to each directory entry. The fields are as follows:

- **`name`**: the name of the directory entry
- **`size`**: the size of the file
- **`firstBlock`**: the first block of the file
- **`type`**: the type of the file casted from the `FilyType` enum
- **`perm`**: the file's read/write/execute permissions
- **`mtime`**: the file's most recent modification time
- **`reserved`**: `16` reserved bytes

Open File Table

We use an **`Open File Table`** linked list to keep track of which files are open. Each element is simply a `File Descriptor`, as described below.

File Descriptor

This is a struct that we used that contains the data pertaining to each file descriptor, which is used in a linked list as our `OFT`, as described above. The fields are as follows:

- **`*de`**: a pointer to the corresponding directory entry
- **`ind`**: the integer (`fd`) assigned to this file descriptor
- **`ref_index`**: the number of `fd` array elements that point to this node

mode: the mode (F_READ, F_WRITE, or F_APPEND)
f_pos: the file offset from the start of the file with respect to the file system
d_pos: the file offset of the directory entry of the file being opened with respect to the file system

Jobs

This is a struct that we created to keep track of the background jobs. The fields are as follows:

job_pid: the pid of the process
job_id: Job index to print in jobs built in command

The related functions are the following:

job* create_job(pid_t pid): creates a job object using the given pid
void free_job(void* j): frees the dynamically allocated pointers in the specified job.
bool job_equal_predicate(void *target_pid, void *curr_job): used as a predicate in a linked_list function that searches for the linked_list_elem with the given pid and the given job
bool job_id_equal_predicate(void *target_job_id, void* curr_job): used as a predicate in a linked_list function that searches for the linked_list_elem with the given job_id and the given job

Linked List

This is a fairly standard generalized linked list. This can be found in linked_list.c, along with the header file linked_list.h .

Extra Credit

Memory Leaks

There should be no memory leaks in the standalone FAT or Penn OS.