PennOS Document for Group 35

Haitian Zhou, Jiayuan Chen, Shuaijiang Liu, Jiahao Huang

Part 1. PennFAT

User Function:

For the file_descriptor, we initialize a linked list which is stored in the PCB so that each thread will have one linked list to store the file_descriptor. All of these file system related system calls are at the user level which is one level below the shell, and are defined inside file_system_call.c.

file_system_call.c:

- f_open(const char *fname, int mode): If the file is opened in the F_WRITE mode, first check there is no other file opened in the writing mode, if so, throw an error. If no, create a new file, or clear all data if the file already existed. Add a new linked list node to the PCB. If the file is opened in the F_READ mode, check if the file already exists and if the file can be read, if not exist or permission not permitted, throw an error. then add a new linked list node to the PCB. If the file is opened in the F_APPEND mode, check if the file exists and the permission, locate the end of file and pass it to the file pointer. Add a new linked list node to the PCB. Throw an error if invalid mode. Return file' fd (which is also its first PennFAT block number) if successful.
- f_read(int fd, int n, char *buf): If the file descriptor is STANDARD IN, call read() method. Else, check the permission and if fd is valid, if not throw an error. Use the while loop to keep read file content from the FAT data region until the limit is reached or EOF has been read. Return how many bytes have been read if successful, -1 otherwise.
- f_write(int fd, const char *str, int n): If the file descriptor is STANDARD OUT, call write() method. Else, check the permission and if fd is valid, throw an error if not. Then keep writing to the buffer until the limit is reached or EOF has been written. If the original file is larger than the current file, remove all exceeding data. Return how many bytes has been written if successful, -1 otherwise.
- f_close(int fd): If file descriptor is 1 or 0, skip. Else, remove the linked list node in the PCB. Return 0 if successful, -1 if invalid fd.
- f_unlink(const char *file_name): Call do_rm() function in PennFAT to remove files in the FAT. Return 0 if successful, -1 if file not found.
- f_lseek(int fd, int offset, int whence): If fd does not exist, throw an error. If whence equals to F_SEEK_CUR, set file descriptor to offset more than original. If whence equals to F_SEEK_END, set file descriptor to the end plus offset. If whence equals F_SEEK_SET, set the file descriptor to offset. Return 0 if successful, -1 if invalid fd or whence.

- f_ls(const char *file_name): If there is no argument, call do_ls() in PennFAT to list all files. Otherwise, print the argument out if the argument file exists.
- f_move(char *src, char *dest): call do_mv() in PennFAT, return 0 if successful, -1 if file does not exist.
- f_chmod(char *mode, char *file): Call do_chmod() in PennFAT.
- f_mount(char *file_system): Call do_mount() in PennFAT. Return 0 if successful, -1 if mmap error.
- f_umount(): Call do_umount() in PennFAT. Return 0 if successful, -1 if mumap error.
- f_execute(char *file_name): Used to handle the synchronous mode. First check if the file exists and the permission. Use a struct which contains the number of commands and specific command lines and arguments. Copy lines of command line by line from the file to the struct and return it for execution.

Part 2. Kernel

For the kernel part, we have three files: kernel.c, user.c, and linked_list.c. In kernel.c, we developed kernel level system call and helper functions. User functions were implemented in user.c. linked_list.c contains the linked list node and helper functions that we use to maintain the priority queue.

Signals:

- S_SIGSTOP 0
- S_SIGCONT 1
- S_SIGTERM 2

Status:

- RUNNING 0
- STOPPED 1
- BLOCKED 2
- ZOMBIED 3
- SIGNALED 4
- ORPHAN 5

Data Structure:

- Queues: It contains 7 linked list (3 ready queues, zombie, block, stop, and all process queue). It is maintained in kernel.c. Each node represents a process in shell and points to its pcb. Initialized and append to ready queue when calling p_spawn.
- Pcb: It contains necessary flags to check process status, and u_context. Initialized in k_process_create.

Kernel.c:

- Pcb* k_process_create(Pcb* parent): the function will malloc a new pcb for the process we are creating and assign initial determination flags in Pcb. The function will return a new pcb for the process we are creating.
- void k_process_kill(Pcb* process, int signal): kill process to signal.
    - Case 1: signal == S_SIGCONT, process will be put to ready based on its priority and set status to running. If process is already in ready queue or in zombie or orphan status, signal will be ignored.
    - Case 2: signal == S_SIGSTOP, process will be put to stopped queue and set status to stopped. If process is already in stopped queue or in zombie or orphan status, signal will be ignored.
    - Case 3: signal == S_SIGTERM, process will be put to zombie queue and set status to zombie. If process is already in zombie queue with status zombie or orphan, signal will be ignored.
- void k_process_cleanup(Pcb* process): the function will find linked list nodes that the pcb belongs to and remove these nodes from queues. Free pcb and all linked list node of the process.

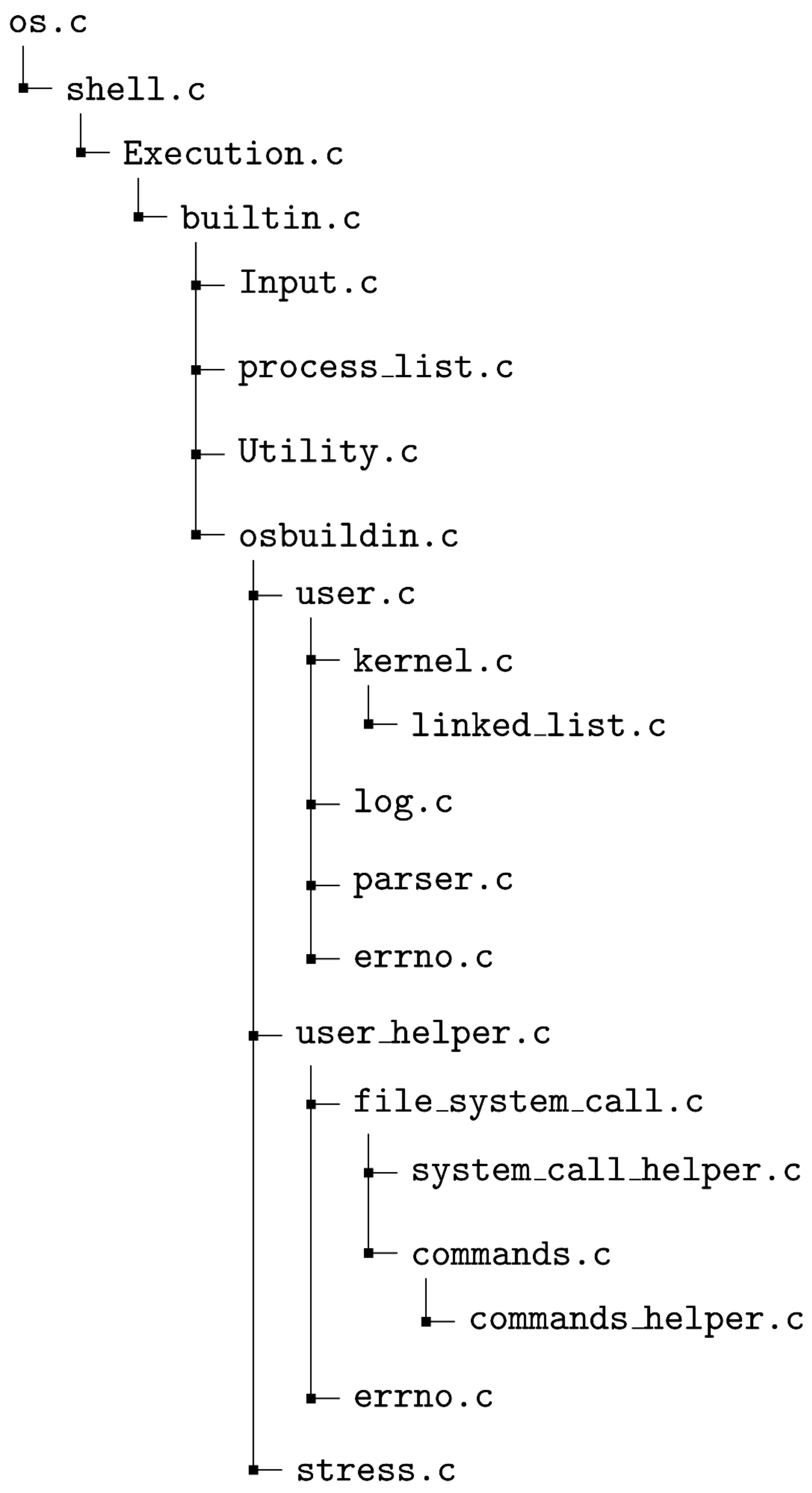
User.c:

- pid_t p_spawn(void (*func) (), char* argv[], int fd0, int fd1): In the function, it calls k_process_create to malloc new pcb, and do make context with u_context in new pcb finally append to default ready queue and all process list. If the function return success, it will return pid of new process. If fail, it will return -1.
- pid_t p_waitpid(pid_t pid, int* wstatus, int nohang): the function will check if desired pid process change status or not. If nothing can be wait or pid not exist, it will return -1 and throw an error. Otherwise, return waited pid. If the waited child is in zombie or orphan status, it will be free.
    - Case 1: pid == -1 and nohang == FALSE. Function will check all children, if anyone change status, it will be waited and its pid will be returned. If no process change status, the thread calling p_waitpid will be blocked and swap to scheduler. The calling thread will be unblocked when any child change status. After it is scheduled, p_waitpid will find out which child change status and return child's pid.
    - Case 2: pid == -1 and nohang == TURE. Function will check all children, if anyone change status, it will be waited and its pid will be returned. If no process change status, function will return -1.
    - Case 3: pid >0 and nohang ==FALSE. Function will find out the pcb of pid. If pid not exist, return -1. By checking pcb, we can know the process change status or not. If changed, process will be waited and return pid. If not changed, the thread calling p_waitpid will be blocked. The calling thread will be unblocked when the pid process change status. After the calling thread is scheduled, it will wait pid process and return pid.

    - Case 4: pid>0 and nohang == TRUE. Function will find out the pcb of pid. If pid not exist, return -1. If pid process change status, return pid, else return -1.
- int p_kill(pid_t pid, int sig): the function will call k_process_kill(). If pid not exist, return -1 and throw an error. Otherwise return 0.
- void p_exit(): free current running thread immediately. If the process is waited, it will be put to zombie queue first. P_waitpid will free it. If it is not waited, it will be free in p_exist().
- int p_nice(pid_t pid, int priority): If pid not exist return -1. Otherwise return 0. Find pcb based on pid than set priority flag to priority. If process is in ready queue, move it to the ready queue of new priority level.
- void p_sleep (unsigned int ticks): calling thread will be set to blocked queue and BLOCKED status. A sleep timer is set up in running thread pcb. It will be deducted and check in scheduler. Last, swapcontext to scheduler_context.

Listing of The Files

The following tree in the next page shows the relationship between each file for Pennos source code files. ‘Parent’ files could include their ‘child’ files. In this tree, our operating system’s system calls (U) are located in user.c and file_system_call.c (i.e. file_system_call.c mainly contains the OS API related to PennFAT).
(Note: all pennos source code is put inside ./src/pennos)



os.c
shell.c
Execution.c
builtin.c
Input.c
process_list.c
Utility.c
osbuildin.c
user.c
kernel.c
linked_list.c
log.c
parser.c
errno.c
user_helper.c
file_system_call.c
system_call_helper.c
commands.c
commands_helper.c
errno.c
stress.c