Companion Document - PennOS

Team 21: Roopa Chandra, Christopher Fisher, Steven Bursztyn, John T. Geyer

Tree:

```
.
├── Makefile
├── README.txt
├── bin
│   ├── fs
│   ├── catFlatFAT
│   ├── lsFlatFAT
│   ├── mkFlatFAT
│   ├── pennos
├── doc
│   ├── CompanionDocument.pdf
├── log
├── obj
│   ├── catFlatFAT.o
│   ├── duplist.o
│   ├── file_directory.o
│   ├── file_system.o
│   ├── ilist.o
│   ├── logger.o
│   ├── lsFlatFAT.o
│   ├── mkFlatFAT.o
│   ├── pennos.o
│   ├── process_table.o
│   ├── schedule.o
│   ├── scheduler.o
│   ├── shell.o
│   ├── shell_aux.o
│   ├── shell_list.o
│   ├── shell_new_jobs.o
│   ├── sleep_queue.o
│   └── tokenizer.o
└── src
    ├── -
    ├── catFlatFAT.c
    ├── duplist.c
    ├── duplist.h
    ├── file_directory.c
    ├── file_directory.h
    ├── file_system.c
```

```
├── file_system.h
├── ilist.c
├── ilist.h
├── logger.c
├── logger.h
├── lsFlatFAT.c
├── macros.h
├── mkFlatFAT.c
├── pennos.c
├── pennos.h
├── process_table.c
├── process_table.h
├── schedule.c
├── scheduler.c
├── scheduler.h
├── shell.c
├── shell.h
├── shell_aux.c
├── shell_aux.h
├── shell_list.c
├── shell_list.h
├── shell_new_jobs.c
├── shell_new_jobs.h
├── sleep_queue.c
├── sleep_queue.h
├── tokenizer.c
└── tokenizer.h
```

---

f_functions:

int f_open(const char * fname, int mode) (U) open a file name fname with the specified mode and return a file descriptor. The allowed modes are as follows: F_WRITE - writing and reading, truncates if the file exists, or creates it if it does not exist; F_READ - open the file for reading *only*, return an error if the file does not exist; F_APPEND - open the file for reading and writing but does not truncate the file if exists; additionally, the file pointer references the end of the file. f_open returns a file descriptor on success and a negative value on error.

int f_read(int fd, char *buf, int n) (U) read n bytes from the file referenced by fd, and copies what it reads to buf. On return, f_read 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) (U) 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) (U) close the file fd and return 0 on success, or a negative value on failure. This system call returns a negative value when the fd passed in is invalid or if the fd does not reference an open file

int f_unlink(const char * fname) (U) remove the file reference by fname, returns negative value on error or 0 on success. It closes the file specified, removes it from the filesystem and frees all file descriptors referencing the file.

int f_lseek(int fd, int offset, int whence) (U) reposition the file pointer for fd to the offset relative to whence. You must also implement the constants F_SEEK_SET, F_SEE_CUR, and F_SEEK END, which reference similar file whences as their similarly named counterparts in lseek(2).

void f_ls() (U) prints out the file list contained in the filesystem. It parses the file system's directory and prints out filename, size and block start number of every file stored in the filesystem.

int f_dup(int fd1, int fd2, int pid) (U) duplicates the first file descriptor (fd1) to be the same as second (fd2). It returns 0 on success and a negative value on a failure. After calling this function, everytime fd1 is used such as in f_read and f_write calls, it refers to fd2 and the file corresponding to fd2.

---

p_functions

int p_spawn_priority(void * func, char *command, char **argv, int priority) spawns off a new process with the specified priority where the process takes in the arguments specified by the (null terminated) array argv

int p_spawn(void *func, char *command, char **argv) spawns off a new process with the default priority where the process takes in the arguments specified by the (null terminated) array argv

int p_exit() stops the execution of the calling process and enters said process into zombie mode. On success will never return.

wait_status_t *p_wait(int mode) when mode=NOHANG, will return immediately if there exists a child process that has changed state. Otherwise, the function will block the calling process until a change occurs in any of the process's children.

wait_status_t *p_wait_pid(int pid, int mode) when mode=NOHANG, will return immediately if there there exists a child process with given pid that has changed state. Otherwise, the function will block the calling process until a change occurs in any of the process's children.

int p_nice(int pid, int priority) changes the priority level of the process with the given pid. Updates the scheduler queue of the process if said process is currently running or in the ready queue.

process_info_t p_info(int pid) returns pid and status information for the process with given pid.

int p_sleep(int ticks) puts the running process to sleep for "ticks" number of ticks (roughly ticks/10 seconds). Returns 0 on success.

int p_get_pid() returns the pid of the current process running.

int p_ps() returns the pid, parent pid, nice, status, and command string for each active process in the process table.

Three Built-In Commands Implemented:

<--> denotes mandatory argument
[---] denotes optional arguments

cp <file1> <file2> copies the contents file1 to file2--it overwrites rather than appends. If file2 doesn't exist, it is created. If file1 doesn't exist, it throws an error. If file1 and file2 are the same, it indicates no change was made. Additional arguments are ignored.

wc [file1] takes in an arbitrary number of files and outputs the newline, word, and byte counts for all the files, or STDIN if no file is provided. The behavior is similar to bash, including when the provided file names don't exist.

rev [file1] reverses each line in file1 or STDIN if no file was provided. Prints an error if the provided file doesn't exist. Additional arguments are ignored.

Compilation Instructions:

Type "make" to first compile the program. Then to run pennos, enter the bin folder and type "./pennos fs" where fs represents the file you wish to use as the filesystem for the operation system. To create an appropriate filesystem, before running "./pennos fs", run "./mkFlatFAT fs" which will initialize the file "fs" as an empty filesystem. You can then provide this file as the argument to pennos.

In addition, typing ./catFlatFAT filesystem test1 -someflag will allow you to interact with the filesystem where the "-someflag" indicates whether you wish to read or write and the test1 indicates the file of interest.

Similarly by typing ./lsFlatFat fs, this will run the lsFlatFat executable which will print out the current state of the filesystem fs.

PCB Structure:

Our PCB structure is defined in the pcb_t struct in "process_table.h". For each process, it stores

- PID
- Priority
- Parent PID
- Child PIDs
- Formatted name of the process
- Status (PT_STOPPED, PT_READY, PT_SLEEPING, PT_RUNNING, PT_ZOMBIE, or PT_WAITING)
- Wait status (corresponding to the last change of the status: WSTAT_EXIT, WSTAT_STOP, WSTAT_CONT, WSTAT_TERM, WSTAT_WAKE, WSTAT_UNCH for unchanged) which is updated whenever the status of the process changes
- Queue of the children it needs to wait on
- Stack id for valgrind debugging
- Open file descriptors
- List of duplicated file descriptors
- ucontext struct

Data Structures and Justifications

Process queue:

We keep a linked list of pcb_t structs, one for every process that has not exited/terminated and been waited on. Functions for modifying this list and the pcb_t's in it are provided in the process_table.h and process_table.c files. We chose to use a linked list since it was relatively easy to implement (as compared to something like a hashtable, or other more optimized data structure) and allowed for dynamic resizing and easy insertion/removal.

Directory file node:

To implement the file system, we take in a file that's been initialized by the mkFlatFAT program. Then, in order to keep track of files, we parse the directory in the file passed in by starting at block 0 and following the directory blocks until reaching an EOF. In each directory block, we first store the num of nodes in that particular directory, with a max of three per block, and then the filename, size of the file and start block of that particular file. We then take this information from the file based in and create a NodeList which is a linked list of node_t's. Each node_t stores this information(filename, size, fstart) which allows us to quickly find crucial information about each file when updating the file system.

Code Layout (in scheduler, kernel, shell, filesystem):

Scheduler:

The scheduler's only job is to output the next pid to be scheduled. It does so by maintaining three queues corresponding to priorities -1, 0, and 1. I use a 3 x 3 matrix to keep track of how many times to choose a pid from each queue. The matrix rows are { 3, 2, 1 }, { 3, 2, 1 }, { 3, 2, 2 }. This just means priority -1 gets scheduled 3 times, then priority 0 gets scheduled twice, then priority 1 gets scheduler once (or twice if this is the third cycle through): this ensures the priority levels are scheduled in a 9:6:4 ratio. This prevents starvation. Moreover, since pids are enqueued at the end of the queues, this also helps prevent starvation. The scheduler.h header file provides methods for adding and removing pids, getting the next pid to be scheduled, and changing the priority of a process.

Kernel**:**

The kernel code is almost all in pennos.c/pennos.h. User functions like p_spawn, p_kill, and p_exit are in general just wrappers for their kernel level equivalents k_process_create, k_process_kill, and k_process_terminate. The kernel also maintains a sleep queue and a

process table to keep track of processes. Basically, k_process_create adds a new pcb_t to the process table and returns the pid of the newly added process. K_process_kill simulates the sending of signals: it adds/removes from the scheduler, adds/removes from the sleep queue, and updates the wait status so that the parent of the process that changes state can be notified. K_process_kill just removes the pid from everywhere except the pcb_t (the pcb_t isn't removed until the process is waited on).

More generally, our kernel relies on user-level functions and kernel functions that interact using a privilege bit--this allows us to execute kernel code if we are in the kernel and to avoid doing so if we are in a user level process. The f_* functions do not use this structure--they just provide a nice interface for the user processes to interface with the file system.

We use an alarm signal to simulate a clock tick, and our alarm signal handler essentially saves the context of the current process, updates the sleep queue (as one "tick" has gone by), and then calls the scheduler function. The scheduler will invoke the scheduler (from the scheduler.h file) to get the next pid. If there are no pids in the scheduler, and idle process is scheduled. Once there is a process to be run, the scheduler sets the context to that process's context.

Shell:

For shell, our code is structured as the following: shell starts off running in the shell() function in shell.c, which prompts the user. Once a command has been entered, it tokenizes it and then calls for a new job to be created and added to the job linked list stored in shell() if it is not a command that should be run as a subprocess in the shell itself. A new job is made in shell_new_jobs().c where we use the tokens to detect redirection and then call p_spawn_priority() to create the new process itself. The list of jobs is implemented in shell_list which supports functionality for updating the job list.

When the process goes to execute, it calls its corresponding function in shell_aux that takes in the arguments initially entered with the command (if there are any) and uses them to execute the command. For example, the remove file command will call a function the remove function in shell_aux that takes in a file and make the appropriate calls to the file system.

Filesystem:

The filesystem is split up into two files -- file_directory.h/c and file_system.h/c -- which handle the directory nodes and the file system, respectively. The filesystem first starts off with the mkFlatFAT command, which initializes the FAT and the directory nodes to show that the filesystem is empty. Then, upon reading/writing/appending/removing a file, you must first always get the directory node list and find the file you're trying to perform an operation on. This gives you the fstart, which you can then follow via links from the directory to find the next part of the file (or EOF -1 if it stops there).

Since we weren't allowed to use -2 to denote free blocks, we had to make an algorithm to find the next free block -- basically we just scan through all files in our directory, follow all chains,

and mark the “seen” blocks in a size 512 array. Then, we choose the first one with entry equal to 0.

We then have some wrapper functions such as write_file_fd that wrap the original functions in file system. This allows us to distinguish between instances when fds are used and when we are running the functions alone for a program such as catFlatFAT. This was necessary to maintain code simplicity.