PennOS

Ethan Chee, Andrew Jiang, Ishaan Lal, Nathaniel Lao

In this document, we outline the files used to construct PennOS as well as describe their functions, utilized data structures, etc.

All System Calls

Kernel: All of these calls exist in kernel-system.c

`Pcb *k_process_create(Pcb* parent);`

Creates a new child thread and associated PCB. The new child PCB will retain all of the same properties as the parent PCB with a few exceptions. Firstly, the new child will have a new process_id and parent_id. Secondly, the new child will have an empty `child_queue`, `zombie_queue`, and `changed_children` queue.

Finally, the new child PCB will have waiting and sleeping set to 0 and finished set to `RUNNING` (see PCB struct for more information on what these variables mean). The function returns a reference to the new child's PCB.

Error conditions:

if `malloc(3)` fails, then it will call `perror(3)` and return. No values of `p_perrno` are set for this system call.

`void k_process_kill(Pcb* Process, int signal);`

Kernel function that sends a signal to a process. Signal can take any of the following values:

`S_SIGSTOP (0)`
: Stops a process. If the process is already stopped nothing will happen.

`S_SIGCONT (1)`
: Continues a process. If the process is already running then nothing will happen.

`S_SIGTERM (2)`
: Terminates a process. This means that the process is exited unconditionally.

This call has no return value.

Error conditions:

If the process cannot be found in the stopped queue when the signal is `S_SIGCONT` or `S_SIGSTOP`, then `p_errno` is set to `p_NOTF`.

`void k_process_cleanup(Pcb* process);`

This function cleans up the memory for a process, removes it from all queues (stopped, blocks, scheduler queues) and kills off all of the children (which would now be orphans). It also removes the process from its parent's queues. If the parent process is not found, then we identify the process as an orphan and terminate early.

This call has no return value nor error conditions.

Utility Functions:

`Pcb *removeFromBlocked(pid_t pid);`

Removes a process from the blocked queue and returns its Pcb (if found).
Sets `p_errno` to `p_NOTF` if not found when expected.

`Pcb *searchBlockedAndStopped(pid_t pid);`

Searches both the blocked and stopped queue for a process.
Returns a pointer to its Pcb if found, or `NULL` if not.

User: All of these calls exist in user.c

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

User function that forks a new thread that retains most of the attributes of the parent thread (see `k_process_create`). Once the thread is spawned, it executes the function referenced by func with its argument array `argv`. `fd0` is the file descriptor for the input file, and `fd1` is the file descriptor for the output file. `foreground` is true if it is meant to be spawned in the foreground, else background (false).

It returns the `pid` of the child thread on success, or -1 on error (if memory cannot be allocated).

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

Very similar to Linux `waitpid(2)`. Sets calling thread to blocked (if `nohang` is false) until the child of calling thread changes state. If `nohang` is true then `p_waitpid` returns immediately. If pid is set to -1 then the calling thread will wait on any of its children. When successful, `p_waitpid` returns the pid of the child that changed states. If `nohang` and no children are waitable then 0 is returned. The function returns -1 if there is an error. For example, if the child referenced by pid cannot be found, `p_waitpid` will return -1.

`int p_kill(pid_t pid, int sig);`

Sends the signal sig to the thread referenced by pid. It returns 0 on success, or -1 on error.
`sig` can take any of the following values:

- `S_SIGSTOP (0)`
  Stops a process.
- `S_SIGCONT (1)`
  Continues a process.
- `S_SIGTERM (2)`
  Terminates a process.

Please refer to the documentation for k_process_kill for more information how kill handles these signals.

`void p_exit(void);`

Exits the current thread unconditionally.

This was not implemented since our code does not actually make use of p_exit(). Instead, code for exiting a function is located in k_process_kill(). Specifically, when a process is sent a S_SIGTERM signal.

File System: all in system.h

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

Returns file descriptor upon success. Returns -1 upon failure. Error conditions: If the inputted mode is F_READ and the inputted filename does not exist, or if the inputted mode is F_WRITE and the file has already been opened in F_WRITE mode before.

```
int f_read(int fd, int n, char *buf);
```
Takes in an integer file descriptor value, an integer n denoting the number of bytes to read, and a char* buffer which is where the data will be stored. If fd points to standard input, and standard input is not redirected, read from terminal; otherwise, find the corresponding file in the file system, start from the current file position, and read n bytes. Increments file pointer by number of bytes read. Returns number of bytes read.

Error conditions: f_read will always return the number of bytes read– this will only differ from n if EOF is reached. If the inputted fd does not exist/was not opened, then -1 will be returned.

```
int f_write(int fd, const char *str, int n);
```
Takes in an integer file descriptor value, a string that is to be written, and an integer n describing the number of bytes to write. If the fd is stdout, and stdout is not redirected, write to terminal; otherwise, find the corresponding file in the file system, start from the current file position, and write n bytes of str. Increments file pointer by number of bytes written.

Returns number of bytes written upon success. Returns -1 upon failure.

Error conditions: If the inputted fd does not exist/was not opened, then -1 will be returned. If the inputted fd was opened in F_READ mode, then -1 will be returned.

```
int f_close(int fd);
```
Decrements reference count for the file when the reference count is not equal to one. If reference count is one, check if the file is marked for deletion (if first character in name is '\2'). If it is, delete the file; otherwise, just remove the fd.
Returns 0 on successful closing of file. Returns -1 on error.

Error conditions: If fd does not exist/was not opened, then -1 will be returned. If a fd <= 2 (either stdin, stdout, or stderr) is inputted, then -1 will be returned.

```
void f_unlink(const char *fname);
```

Takes in filename and checks whether the file is currently open (if there's an existing file descriptor pointing to that file). If there is, go in the directory file and mark the name with '\2'; otherwise, clear the file and mark the name with '\1'.

This function has no return value or error conditions.

```
void f_lseek(int fd, int offset, int whence);
```

Takes in file descriptor, offset used to set/move pointer, and condition of lseek behavior. The position of the file in the file directory which matches the specified file descriptor has its position updated (in regards to the parameter offset) on having F_SEEK_SET,F_SEEK_CUR or F_SEEK_END for whence.

This function has no return value or error conditions.

```
char* f_ls(const char *fileName);
```

Upon NULL fileName, operates as normal ls would by accessing all non-deleted files in the file system, extracting attributes about these files (such as first block, permissions, size. etc) and compiling them into a string to be printed. Note that many of these attributes appear in the directoryEntry data structure. For a non-NULL fileName, only extracts attributes about the specified file.
Obtaining/extraction occurs by lseeking to appropriate region determined by fileName and reading entry information.

Returns a char* which contains a string representation of the ls output.
Error conditions: This function has no error conditions.

```
void f_rename(const char* oldName, const char* newName);
```

System call to rename a file. Parameters include the oldName which is the file to be renamed as well as the newName. Implementation involves lseeking to the file found by oldName and overwriting name.
This function returns nothing and has no error conditions.

`void f_chmod(const char* fileName, int mode);`
Analyzes mode to determine which permissions to add or subtract. lseeks to read the current permissions of the file, and updates them accordingly by rewriting at that spot.
This function returns nothing and has no error conditions.

Perror: found in perror.h and perror.c

`void reset_p_errno();`
Resets the global `p_errno` variable to 0.

`void p_perror(char *s);`
Similar to `perror(3)`. Has the following values:

- `p_INVALID`
  Invalid command passed in
- `p_NOTF`
  Process not found when expected
- `p_PNOTF`
  Parent not found where expected
- `p_CNOTF`
  Child not found where expected
- `p_ARGS`
  Wrong number of arguments for a given shell command
- `p_FDINV`
  File descriptor does not exist
- `p_PERMS`
  File descriptor does not have correct permissions
- `p_FDWRITE`
  File descriptor already opened in write mode
- `p_FILEINV`
  File does not exist
- `p_INVCLOSE`
  Tried to close `stderr`, `stdin`, or `stdout`

Signals

`S_SIGSTOP`

Stop a process (Ctrl + Z).

`S_SIGCONT`

Continues a process.

`S_SIGTERM`

Terminates a process.

Data Structures

Kernel

`Pcb`:

The structure for a PCB can be found in pcb.h.

Every PCB struct contains the following:

`int process_id;`

pid of the process associated with this pcb

`pid_t parent_id;`

pid of the parent of the process associated with this pcb

`int f_stdin;`

File descriptor of stdin

`int f_stdout;`

File descriptor of stdout

`char* process_name;`

Name of the builtin/command that the process is associated with

`int priority;`

Priority of the process. Priority can either be -1, 0, or 1. Processes with priority -1 will run 1.5 times more than priority 0 which will run 1.5 times more than 1.

`struct Queue* zombie_queue;`

Queue of processes who are children of the process associated with this PCB and who are zombies.

`struct Queue* changed_children;`

Queue of processes who are children of the process associated with this PCB who have changed statuses (i.e. running -> stop or stop -> running)

`struct Queue* child_queue;`

Queue of all processes who are children of the process associated with this PCB

`struct ucontext_t* context;`

ucontext_t data structure associated with this process.

`int waiting;`

integer representing the pid of the process that this process is waiting on. If the process is waiting for any child then waiting will be -1.

`int sleeping;`

If the process is sleeping then this integer will be set to the clock tick at which the process will no longer be sleeping.

`int finished;`

Integer representing the state of this process. It can either be set to EXITED(-1), STOPPED(0), RUNNING(1), SIGNALED(2).

`Queue`:

Found in queue.h

Contains:

`struct QueueNode *head;`

Head of the queue

`struct QueueNode *tail;`

Tail of the queue

`int size;`

Number of QueueNodes

Operations:

`void init(struct Queue *queue);`

Initializes queue struct.

`void enqueue(struct Queue *queue, struct Pcb *pcb, pid_t pid);`

Adds a new node to the queue.

`struct Pcb *removeQueueNode(struct Queue *queue, pid_t pid);`

Removes a node given a pid, returns the pcb. Returns NULL if empty.

`struct Pcb *dequeue(struct Queue *queue);`

Removes the head and returns the pcb. Returns NULL if empty.

`struct Pcb *findQueueNode(struct Queue *queue, pid_t pid);`

Linear search for a pid. Returns NULL if not found.

`int size(struct Queue *queue);`

Returns size of the queue

`void printQueueProcesses(struct Queue *queue, char status);`

Used in implementing the `ps` shell command. Prints queue in the correct format with the given status as the 4th column.

`QueueNode`:

Found in queue.h

Contains:

`pid_t pid;`

Pid of the node

`struct QueueNode *next;`

Pointer to the next node in the linked list

`struct Pcb *Pcb;`

Pcb of this node

job_queue:

Found in job_handler.h

Contains:

`int jobId;`

Job id, in accordance to the spec

`int pid;`

Pid of the process that this node contains

`char* command;`

Command string for printing

`int status;`

Status of the job

`int pid_status;`

Status of the process

`struct job_queue_struct* next;`

Next job in the queue

Operations:

```
int add_to_job_queue(job_queue** head, pid_t pid, char*
        command);
```

Adds job to job queue. Returns -1 if failed, otherwise jobId of the newly created job.

`void update_status(job_queue** head, pid_t pid, int status);`

Updates the status of a job with the given pid to the given status

`void return_job_queue(job_queue** head);`

Prints the required string for `jobs`

`void delete_job(job_queue** head, pid_t pid);`

Removes the job with the given pid from the job_queue

`void delete_queue(job_queue** head);`

Destructs the queue

`int get_current_job(job_queue** head);`

Returns pid of current job.

`int get_num_jobs(job_queue** head);`

Returns the size of the job queue.

`char* get_name(job_queue** head, pid_t pid);`
Gets string name of a job for printing purposes
`int get_status(job_queue** head, pid_t pid);`
Returns status of a given job.
`void terminate_process(job_queue** head, pid_t pid);`
Exits a process in the job queue.
`int clean_dead_jobs(job_queue** head);`
Check if there are any 'done' jobs, and returns 1 if one was removed, 0 otherwise.
`int get_job_by_id(job_queue** head, int job_id);`
Returns pid of a given job. Returns 0 if not found.

File System

`Fatll:`
Linked list data structure that is often used when working with the FAT. Commonly used to get all block numbers associated with a single file.
`int index;`
Value of block to look at.
`struct fatll *next;`
Pointer to next node.

`filell:`
`int fd;`
Integer value representing the file descriptor.
`int refCount;`
Reference count value. Increments by one if calling f_open on the same file in the same mode.
`int mode;`
Denotes file mode such as F_READ, F_APPEND, etc.
`int position;`
Denotes current position in the file.
`int offset;`
Offset indicating where in the file system the directory entry for the current file is.
`struct filell *next;`
Pointer to next file descriptor
`struct filell *prev;`
Pointer to previous file descriptor

`directoryEntry:`
`char name[32];`
null-terminated file name.

```
        uint32_t size;
          number of bytes in file
        uint16_t firstBlock;
          first block number of file. undefined if size = 0
        uint8_t type;
          0 = unknown, 1 = regular, 2 = directory file, 4 = symbolic
          link
         uint8_t perm;
          0 = none, 2 = write, 4 = read, 5 = read & exec, 6 = read &
          write, 7 = read, write, exec
         time_t mtime;
          creation/modification time (use time(2)s)

    fileSystem:
      uint16_t *fat;
          List marking the start of the fat region.
      uint16_t *dataRegion;
          List marking the start of the data region.
```

File Structure

```
|-- Makefile
|-- README.md
|-- bin
|-- doc
|-- log
|   `-- log
|-- src
|   |-- fs
|   |   |-- fatll.h
|   |   |-- filell.h
|   |   |-- inputParse.c
|   |   |-- inputParse.h
|   |   |-- pennfat.c
|   |   |-- pennfat.h
|   |   |-- shell.c
|   |   |-- system.c
|   |   `-- system.h
|   |-- kernel
|   |   |-- builtins.c
|   |   |-- builtins.h
|   |   |-- job_handler.c
```

```
|   |   |-- job_handler.h
|   |   |-- kernel-system.c
|   |   |-- kernel-system.h
|   |   |-- kernel.c
|   |   |-- kernel.h
|   |   |-- log.c
|   |   |-- log.h
|   |   |-- parser-aarch64.o
|   |   |-- parser-x86_64.o
|   |   |-- parser.h
|   |   |-- pcb.h
|   |   |-- penn-shell.c
|   |   |-- penn-shell.h
|   |   |-- pipe.c
|   |   |-- pipe.h
|   |   |-- queue.c
|   |   |-- queue.h
|   |   |-- scheduler.c
|   |   |-- scheduler.h
|   |   |-- stress.c
|   |   |-- stress.h
|   |   |-- user.c
|   |   `-- user.h
|   |-- macros.h
|   |-- p_errno.c
|   `-- p_errno.h
`
```