Penn OS Companion Document: Group 19

NOTE: PENNFAT IS AT THE END

NOTE: NEW FILE DEFINED ON EACH NEW PAGE

pennos.c

void setTimerAlarmHandler(void)

Input Arguments: nothing

Return Values: No return values

int main(int argc, char argv)**

Input Arguments: number of arguments, array of arguments in strings

Return Values: returns 0

dependencies.h

```
#pragma once

#include <stdio.h>

// Define the structure for a Process

typedef struct process{

   struct pcb* pcb;

   struct process* next;

} Process;
```

Declare the heads and tails of all the queues being used (high, medium, low, blocked, stopped, zombie, orphan, temp

```
extern Process *highQhead;

extern Process *highQtail;

extern Process *medQhead;

extern Process *medQtail;

extern Process *lowQhead;

extern Process *lowQtail;

extern Process *blockedQhead;

extern Process *blockedQtail;

extern Process *stoppedQhead;

extern Process *stoppedQtail;

extern Process *zombieQhead;

extern Process *zombieQtail;

extern Process *orphanQhead;

extern Process *orphanQtail;

extern Process *tempHead;

extern Process *tempTail;

extern int ticks;

extern int fgpid;

static const int quantum = 100000;
```

kernel.h

```
#pragma once
#include <ucontext.h> // getcontext, makecontext, setcontext, swapcontext
#include "pcb.h"
#include "scheduler.h"
#include "shell.h"
#include "user.h"
#include "dependencies.h"

#define S_SIGTERM 1
#define S_SIGSTOP 2
#define S_SIGCONT 3
struct pcb* k_process_create(struct pcb *parent);
Input Arguments: PCB struct of the parent process
Return Values: Returns the PCB of the child that was created

void k_process_cleanup(Process *p);
Input Arguments: Process p that is to be cleaned up
Return Values: No return values

int k_process_kill(Process *p, int signal);
Input Arguments: Process p to be killed, signal value of signal
Return Values: returns success or failure value

Process *findProcessByPid(int pid);
Input Arguments:  pid of process to find
Return Values: returns process that has been found
```

pcb

```
#pragma once

#include <ucontext.h> // getcontext, makecontext, setcontext, swapcontext
#include <stdio.h>
#include <stdlib.h>
#include "user.h"
#include "../fs/user.h"

Status values of the various processes
#define ZOMBIE 5
#define BLOCKED 4
#define STOPPED 3
#define RUNNING 2
#define SIG_TERMINATED 1
#define TERMINATED 0
Foreground and Background definitions
#define FG 0
#define BG 1
#define MAX_FILES 512
extern int pidCounter; // keeps track of pid to give to new processes
struct pcb {
   ucontext_t context; // stores the context
   int jobID; // stores the pennshell job ID
   int numChild; // stores the number of children
   int pid; // stores the pid
   int ppid; // stores the parent pid
   int waitChild; // stores pid of the child being waited on currently
   int priority; // stores the priority of the process as an integer (0, 1, -1)
   char *argument; // stores the argument of the process
   int status; // stores the current status of the process
```

```
    int bgFlag; // stores the background or foreground status of the process
    int *childPids;  // list of all pids in the job
    int *childPidsFinished;  // boolean array list that checks every pid is finished
    int sleep_time_remaining; // number of ticks left for sleep to terminate gracefully
    int changedStatus; // whether or not the process has changed status after creation or last wait
    file_t *fd_table[MAX_FILES]; // fd table
};
```

```
char* strCopy(char* src, char* dest);
```

Input Arguments: source string, destination string

Return Values: destination string where the original one has been copied

```
struct pcb *initPCB();
```

Input Arguments: Nothing

Return Values: Struct pcb of the penn shell

```
struct pcb *createPcb(ucontext_t context, int pid, int ppid, int priority, int status);
```

Input Arguments: Context of the process, pid of the process, parents pid, priority of process, status process should start in

Return Values: Struct pcb of the new process

```
void freePcb(struct pcb *pcb_obj);
```

Input Arguments: pcb of the target process

Return Values: nothing

scheduler.h

```
#pragma once
#include <signal.h>
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <sys/time.h>
#include <ucontext.h>
#include <unistd.h>
#include <valgrind/valgrind.h>
#include "pcb.h"
#include "kernel.h"
#include "user.h"
#include "shell.h"
#include "user_functions.h"
#include "dependencies.h"

// high, medium and low priority definitions
#define PRIORITY_HIGH -1
#define PRIORITY_MED 0
#define PRIORITY_LOW 1

extern ucontext_t schedulerContext; // context to return to scheduler
extern ucontext_t *activeContext; // context to run the active process
extern ucontext_t idleContext; // context to run the idle process
extern ucontext_t terminateContext; // context to run the terminate process function

void terminateProcess(void);
Input Arguments: nothing
Return Values: nothing
```

void scheduler(void);

Input Arguments: nothing

Return Values: nothing

void initContext(void);

Input Arguments: nothing

Return Values: nothing

void enqueueBlocked(Process* newProcess);

Input Arguments: Process struct of process to be enqueued

Return Values: nothing

void enqueueStopped(Process* newProcess);

Input Arguments: Process struct of process to be enqueued

Return Values: nothing

void enqueue(Process* newProcess);

Input Arguments: Process struct of process to be enqueued

Return Values: nothing

void enqueueZombie(Process* newProcess);

Input Arguments: Process struct of process to be enqueued

Return Values: nothing

void dequeueZombie(Process* newProcess);

Input Arguments: Process struct of process to be dequeued

Return Values: nothing

void dequeueBlocked(Process* newProcess);

Input Arguments: Process struct of process to be dequeued

Return Values: nothing

void dequeueStopped(Process* newProcess);

Input Arguments: Process struct of process to be dequeued

Return Values: nothing

void dequeue(Process* newProcess);

Input Arguments: Process struct of process to be dequeued

Return Values: nothing

void iterateQueue(Process *head);

Input Arguments: Process struct of head of the queue to be iterated through

Return Values: nothing

void alarmHandler();

Input Arguments: nothing

Return Values: nothing

void setTimer(void);

Input Arguments: nothing

Return Values: nothing

void freeStacks(struct pcb *p);

Input Arguments: pcb struct of process stack to be freed

Return Values: nothing

shell.h

```
#pragma once
#include <signal.h> // sigaction, sigemptyset, sigfillset, signal
#include <stdio.h> // dprintf, fputs, perror
#include <stdbool.h> // boolean
#include <stdlib.h> // malloc, free
#include <sys/time.h> // setitimer
#include <ucontext.h> // getcontext, makecontext, setcontext, swapcontext
#include <unistd.h> // read, usleep, write
#include <valgrind/valgrind.h>
#include <sys/wait.h>
#include <fcntl.h>
#include "kernel.h"
#include "dependencies.h"
#include "user_functions.h"
#include "parser.h"
#include "user.h"

#define INPUT_SIZE 4096 // maximum input buffer size

// process status definitions
#define STOPPED 3
#define RUNNING 2
#define SIG_TERMINATED 1
#define TERMINATED 0
// process foreground or background definitions
#define FG 0
#define BG 1
// define true or false
#define TRUE 1
#define FALSE 0
```

```
// defined the signals
#define S_SIGTERM 1
#define S_SIGSTOP 2
#define S_SIGCONT 3

extern int IS_BG; // global variable to state the status of the current active process

struct Job{
   int myPid; // job ID
   int JobNumber; // Counter for current job number since first job begins from 1
   int bgFlag; // FG = 0 and BG = 1
   struct Job *next; // pointer to next job
   char *commandInput; // argument
   int status; // tell whether its running or stopped
   int *pids; // list of pids of children
   int numChild; // number of children
   int *pids_finished; // boolean array list that checks every pid is finished
};

void setTimer(void);
Input Arguments: nothing
Return Values: nothing

void signalHandler(int signal);
Input Arguments: defined value of the signal
Return Values: nothing

void sigIntTermHandler(int signal);
Input Arguments: defined value of the signal
Return Values: nothing
```

void sigcontHandler(int signal);

Input Arguments: defined value of the signal

Return Values: nothing

void sigtstpHandler(int signal);

Input Arguments: defined value of the signal

Return Values: nothing

void setSignalHandler(void);

Input Arguments: nothing

Return Values: nothing

void pennShredder(char* buffer);

Input Arguments: input buffer

Return Values: nothing

void pennShell();

Input Arguments: nothing

Return Values: nothing

struct Job *createJob(int pid, int bgFlag, int numChildren, char *input);

Input Arguments: pid of job, bg or fg flag, number of children, and input array

Return Values: Job struct

struct Job *addJob(struct Job *head, struct Job *newJob);

Input Arguments: head of the jobs linked list in shell, job to be added

Return Values: Job struct of head of linked list in shell

struct Job *removeJob(struct Job *head, int jobNum);

Input Arguments: head of the jobs linked list in shell, job number of job to be removed

Return Values: Job struct of head of linked list in shell

struct Job *getJob(struct Job *head, int jobNum);

Input Arguments: head of the jobs linked list in shell, job number of job we are finding

Return Values: Job struct of job to be found

int getCurrentJob(struct Job *head);

Input Arguments: head of the jobs linked list in shell

Return Values: job number of chosen job to be returned

void changeStatus(struct Job *head, int jobNum, int newStatus);

Input Arguments: head of the jobs linked list in shell, job number of job whos status is to be changed, new status value

Return Values: Job struct of head of linked list in shell

void changeFGBG(struct Job *head, int jobNum, int newFGBG);

Input Arguments: head of the jobs linked list in shell, job number of job whos status is to be changed, new status value

Return Values: nothing

void iterateShell(struct Job *head);

Input Arguments: head of the jobs linked list in shell

Return Values: nothing

user_functions.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <dirent.h>
#include <sys/stat.h>
#include <time.h>
#include <pwd.h>
#include <grp.h>
#include <ucontext.h>
#include "scheduler.h"
#include "dependencies.h"

void echoFunc(int argc, char *argv[]);
Input Arguments: number of arguments, array of strings of arguments
Return Values: nothing

void sleepFunc(int argc, char *argv[]);
Input Arguments: number of arguments, array of strings of arguments
Return Values: nothing

void busyFunc(void);
Input Arguments: nothing
Return Values: nothing

void idleFunc();
Input Arguments: nothing
Return Values: nothing
```

// ==== filesystem ====

void catFunc(int argc, char **argv);

Input Arguments: number of arguments, array of strings of arguments

Return Values: nothing

void lsFunc(int argc, char **argv);

Input Arguments: number of arguments, array of strings of arguments

Return Values: nothing

void touchFunc(int argc, char **argv);

Input Arguments: number of arguments, array of strings of arguments

Return Values: nothing

void mvFunc(int argc, char **argv);

Input Arguments: number of arguments, array of strings of arguments

Return Values: nothing

void cpFunc(int argc, char **argv);

Input Arguments: number of arguments, array of strings of arguments

Return Values: nothing

void rmFunc(int argc, char **argv);

Input Arguments: number of arguments, array of strings of arguments

Return Values: nothing

void chmodFunc(int argc, char **argv);

Input Arguments: number of arguments, array of strings of arguments

Return Values: nothing

void psFunc (int argc, char **argv);

Input Arguments: number of arguments, array of strings of arguments

Return Values: nothing

void killFunc (int argc, char **argv);

Input Arguments: number of arguments, array of strings of arguments

Return Values: nothing

void man();

Input Arguments: nothing

Return Values: nothing

void zombify(int argc, char **argv);

Input Arguments: number of arguments, array of strings of arguments

Return Values: nothing

void zombie_child();

Input Arguments: nothing

Return Values: nothing

void orphanify(int argc, char **argv);

Input Arguments: number of arguments, array of strings of arguments

Return Values: nothing

void orphan_child();

Input Arguments: nothing

Return Values: nothing

void niceFunc(char *argv[]);

Input Arguments: number of arguments, array of strings of arguments

Return Values: nothing

int nice_pid(char *argv[]);

Input Arguments: array of strings of arguments

Return Values: nothing

void logout();

Input Arguments: nothing

Return Values: nothing

user.h

```
#pragma once

#include <stdio.h>
#include <string.h>
#include <ucontext.h>
#include "scheduler.h"
#include "pcb.h"
#include "parser.h"
#include "dependencies.h"

#define MAX_CMD_LENGTH 1000 // maximum possible length of commands
#define MAX_ARGS 10 // maximum possible number of arguments

// definitions of the signals
#define S_SIGTERM 1
#define S_SIGSTOP 2
#define S_SIGCONT 3

// global variable for actively running process
extern Process *activeProcess;

#define PROMPT "$ " // prompt definition
#define BUFFERSIZE 4096 // maximum size of buffer

char* concat(int argc, char *argv[]);
Input Arguments: number of arguments, array of arguments in form of strings
Return Values: returns concatenated string

pid_t p_spawn(void (*func)(), char *argv[], int fd0, int fd1);
Input Arguments: function to be run, array of arguments, input and output file descriptors
```

Return Values: pid of newly p_spawned process

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

Input Arguments: pid of process to be waited on, new status of process waited on, hang or no hanging wait

Return Values: pid of process that was waited on

void p_sleep(unsigned int ticks1);

Input Arguments: number of ticks left in running

Return Values: nothing

int p_kill(pid_t pid, int sig);

Input Arguments: pid of process to be killed, signal value in int

Return Values: -1 on failure, or return value of k_process_kill

void p_exit(void);

Input Arguments: nothing

Return Values: nothing

int p_nice(pid_t pid, int priority);

Input Arguments: pid of process priority to be changed, new priority

Return Values: returns 0 or -1 based on passing or failure

Filesystem Working Notes

File layout:

- src/
  - pennfat.c - entrypoint for the pennfat standalone
  - fs/
    - fat.c/.h - kernel-level fat functions, file struct defs
    - user.c/.h - user-level fat functions

Namespacing

All kernel-level functions will start with the prefix `fs_`.

All user-level functions will start with the prefix `f_`.

Error Codes

PennOS specific filesystem errors will be in the 1000-2000 range.

```c
#define PEHOSTFS 1001  // could not open/close file in host filesystem
#define PEHOSTIO 1002  // could not perform I/O in host filesystem
#define PEBADFS 1003   // invalid PennFAT file, or was otherwise unable to mount
#define PENOFILE 1004  // specified file does not exist
#define PEINUSE 1005   // the specified file is in use in another context and an exclusive operation was called
#define PETOOFAT 1006  // the filesystem is too fat and has no space for a new file
#define PEFMODE 1007   // attempted operation on a file in the wrong mode
#define PEFPERM 1008   // attempted operation on a file without read/write permissions
#define PEFNAME 1009   // the filename is invalid
```

```
#define PETOOMANYF 1101 // you have too many files open already
#define PESTDIO 1102    // tried to read from stdout or write to stdin
```

Structs

When a file is opened, it returns a `struct file`:

```c
typedef struct file {
    filestat_t *entry;  // mmaped to file entry in directory
    uint32_t offset;  // current seek position
    int mode;
    uint8_t stdiomode;  // 0 = FAT file, 1 = stdout, 2 = stdin
} file_t;
```

where `filestat_t` is the directory entry defined in the PennOS handout:

```c
typedef struct filestat {
    char name[32];
    uint32_t size;
    uint16_t blockno;
    uint8_t type;
    uint8_t perm;
    time_t mtime;
    uint8_t unused[16];
} filestat_t;
```

The OS should maintain a file descriptor table for each process linking an int to one of these `struct file`s.

The low-level filesystem implementation operates using pointers to open file structs.

stdin/stdout

Each `file_t *` struct contains information on whether it is a special file for reading from/writing to stdin/stdout.

The user-level functions `f_read()` and `f_write()` should check this information and redirect to the C API where

necessary rather than the FAT API.

Each process, on creation, should set entries `PSTDIN_FILENO` and `PSTDOUT_FILENO` in its PCB's fd table to file structs

with the correct flag set. This allows for later redirecting stdin/stdout by overwriting the entries in the fd table

with file structs linked to files on the FAT filesystem.

File Locking Mechanism

In order to prevent multiple writers/conflicting read-writes, the PennFAT filesystem grants exclusive access to any

process that opens a file in a writing mode, and shared access to processes opening a file in read mode. To accomplish

this, the filesystem keeps a record of what files have been opened and in what mode; if a call to `fs_open()` would

violate the locking semantics, the syscall fails with an error.

This record is a `file_t *` array that utilizes array doubling to grow dynamically (initially sized at 4).

Standalone

The standalone completes the demo plan with no "definitely/indirectly/possibly lost" memory leaks.

Syscalls

int f_open(const char *fname, int mode)

Open a file. If the file is opened in F_WRITE or F_APPEND mode, the file is created if it does not exist.

Parameters

- `name`: the name of the file to open
- `mode`: the mode to open the file in (F_WRITE, F_READ, or F_APPEND).

Returns

the file descriptor of the opened file, -1 on error

Exceptions

- `PENOFILE`: the requested file was in read mode and does not exist
- `PEHOSTIO`: failed to read from/write to host filesystem
- `PETOOFAT`: the operation would make a new file but the filesystem is full
- `PEFNAME`: the operation would make a new file but the filename is invalid
- `PEINUSE`: the requested file was opened in an exclusive mode and is currently in use

int f_close(int fd)

Closes the specified file, freeing any associated memory.

Parameters

- `fd`: the file to close

Returns

0 on a success, -1 on error.

Exceptions

- `PEINVAL`: the file descriptor is invalid

ssize_t f_read(int fd, int n, char *buf)

Read up to `n` bytes from the specified file into `buf`.

Parameters

- `fd`: the file to read from
- `n`: the maximum number of bytes to read
- `buf`: a buffer to store the read bytes

Returns

the number of bytes read; -1 on error

Exceptions

- `PEFPERM`: you do not have permission to read this file
- `PEHOSTIO`: failed to read from host filesystem

ssize_t f_write(int fd, const char *str, ssize_t n)

Write up to `n` bytes from `buf` into the specified file.

Parameters

- `fd`: the file to write to
- `str`: a buffer storing the bytes to write
- `b`: the maximum number of bytes to write

Returns

the number of bytes written; -1 on error

Exceptions

- `PEFMODE`: the file is not in write or append mode

- `PEFPERM`: you do not have permission to write to this file

- `PEHOSTIO`: failed to read from host filesystem

- `PETOOFAT`: filesystem is full

int f_unlink(const char *fname)

Removes the file with the given name.

Parameters

- `fname`: the name of the file to delete

Returns

0 on success; -1 on error

Exceptions

- `PENOFILE`: the specified file does not exist

- `PEHOSTIO`: failed to i/o with the host entry

uint32_t f_lseek(int fd, int offset, int whence)

Seek the file offset to the given position, given by `offset`. If `whence` is `F_SEEK_SET` this is relative to the

start of the file, for `F_SEEK_CUR` relative to the current position, and for `F_SEEK_END` relative to the end of the

file.

Parameters

- `fd`: the file to seek

- `offset`: where to seek to relative to `whence`

- `whence`: the seek mode

Returns

the new location in bytes from start of file; -1 on error

Exceptions

- `PEINVAL`: whence is not a valid option

filestat_t **f_ls(const char *fname)

Gets information for a file. If `fname` is NULL, gets information for all the files.

It is the caller's responsibility to free each of the returned structs. Use the convenience function `f_freels()` to

do this quickly.

Parameters

- `fname`: the name of the file to get the stat of, or NULL to list all files

Returns

a pointer to an array of filestat struct pointers. The array will always be terminated with a NULL pointer.

Exceptions

- `PEHOSTIO`: failed to read from host filesystem

void f_freels(filestat_t **stat)

Free the filestat list returned by `f_ls()`.

int f_rename(const char *oldname, const char *newname)

Rename a file.

Parameters

- `oldname`: the old name

- `newname`: the new name

Returns

0 on success, -1 on failure

Exceptions

- `PENOFILE`: the file does not exist

- `PEHOSTIO`: failed to perform IO on host drive

- `PEFNAME`: the new name is invalid

int f_chmod(int fd, char mode, uint8_t bitset)

Edit the I/O permissions of an open file.

Parameters

- `fd`: the file whose permissions to edit

- `mode`: the mode to edit it in; '+', '=', or '-'

- `bitset`: the permission bitset to edit by (a combination of FAT_EXECUTE, FAT_WRITE, and FAT_READ)

Returns

0 on success, -1 on error

Exceptions

- `PEINVAL`: the mode is invalid

Kernel-Level Functions

Kernel-level functions are documented inline in src/fs/fat.c.