CS 446/646 – Principles of Operating Systems

Homework 5

Due date: Thursday, 11/30/2023, 11:59 pm

Objectives: You will implement certain functionalities of a simpler (and simulated) version of the ext2 Filesystem. You will be able to describe what inodes are, and how an inode-based Filesystem is structured and its main functionalities. You will be able to describe how Bitmap data structures facilitate faster file / directory creation and Data Block allocation.

General Instructions & Hints:

- All work should be your own.
- The code for this assignment must be in C. Global variables are not allowed except any explicitly specified ones, and you must use function declarations (prototypes). Name files exactly as described in the documentation below. All functions should match the assignment descriptions. If instructions about parameters or return values are not given, you can use whatever parameters or return value you would like.
- All work must compile on **Ubuntu 18.04**. Use a **`Makefile`** to control the compilation of your code. The **`Makefile`** should have at least a default target that builds your application.
- To turn in, create a folder named **`PA5_Lastname_Firstname`** and store your **`Makefile`**, and your **`.c`** source code files in it. Do not include any executables. Then compress the folder into a **`.zip`** or **`.tar.gz`** file with the same filename as the folder, and submit that on WebCampus.

Background:

ext2 Filesystem

The *2nd Extended Filesystem* (https://tldp.org/LDP/tlk/fs/filesystem.html) is laid out as a sequence of Block Groups, within each of which there exists a Superblock, a Group Descriptors Table, a Blocks Bitmap, an Inodes Bitmap, the Inode Table, and finally the Data Blocks:

a) **Superblock**: This data structure contains information about the size and shape (e.g. number of Blocks per Group) of the Filesystem. The same Superblock information is copied in the beginning of every separate Block Group, for redundancy.

The Superblock takes up exactly 1 Block.

b) **Group Descriptors Table**: A single Group Descriptor data structure contains information about the Disk location of the data that describe a specific Block Group, as well as some additional data. The Group Descriptors Table is the array of all Group Descriptors for all Block Groups. This is also copied in the beginning of every separate Block Group, for redundancy.

The Group Descriptors Table takes up an integral (≥ 1) amount of Blocks inside the Group.

c) **Blocks Bitmap**: A simple bitmap for storing the state (free=0 / occupied=1) of the Data Blocks of this Block Group (inside its Data Blocks region).

The Blocks Bitmap takes up exactly 1 Block inside the Group.

d) **Inodes Bitmap**: A simple bitmap for storing the state (unused=0 / used=1) of the inodes of this Block Group (inside its Inodes Table region).

The Inodes Bitmap takes up exactly 1 Block inside the Group.

e) **Inodes Table**: An inode is a data structure that holds information about a regular file / a directory (other types as well, such as character / block device, etc.). Such information is for example the location on the Disk of the regular file's data (stored within some Blocks of the Data Blocks region) / the directory's Directory Entries (also stored within some Blocks of the Data Blocks region). The inodes Table is the array of all inodes for this Block Group. This is allocated at the time of creation of the Filesystem, and therefore the number of inodes is fixed. The state of these inodes (unused/used) is tracked by the Inodes Bitmap.

The Inodes Table takes up an integral (≥ 1) amount of Blocks inside the Group.

f) **Data Blocks**: A Data Block can be used to store a file's data (all or part of it, if the file cannot fit inside a single Block) / a directory's Directory Entries (all or part of them, if they cannot all fit inside a single Block). The Data Blocks region is an array of Blocks that can be used for these purposes. The state of each Data Block (free/occupied) is tracked by the Blocks Bitmap.

The Data Blocks take up an integral amount of N Blocks inside the Group.

Read-Modify-Write
To modify the persistent storage information on the Disk, we perform read / write cycles. We first read-in to Memory (from Disk) the data structures we intend to modify, then perform the necessary modifications, and then write-them-out to Disk (from Memory).

Inodes
Some important things to know about inodes:

i) Part of an inode's data structure is an array of 15 Data Block locations (on the Disk). The first 12 are "Direct Blocks", i.e. the Blocks themselves store whatever data the file/directory holds. The 13th one is a "Single-Indirect Block", i.e. it stores the locations of other Blocks (**`BLKSIZE/ADDRSIZE`**) that they themselves hold actual data. Then the 14th one is a "Double-Indirect" and the 15th one a "Triple-Indirect" Block (the premise is similar as before, just adding 1 more level of indirection at each one)

ii) The Filesystem Root inode is number 2. The inode number 1 is reserved for a special purpose, and that is to store bad block information for the Group. There is no inode number 0, "0" in terms of inodes has the meaning of an invalid value (like NULL).

iii) An inode's type (file / folder) is not stored within the inode data structure, but within the Directory Entry that refers to it.

Directory Entries
Some important things to know about Directory Entries:

i) A Directory Entry corresponds to an inode. It therefore holds the number of that inode, and its type (file/directory). It also holds its human-readable name and its name-length. For the ext2 Filesystem, the name character array is variable in length.

ii) A Directory (i.e. a directory-type inode) holds instead of data, a sequence of Directory Entries (i.e. other inodes that can be files or directories themselves). As mentioned above, these are stored in the Directory inode's Data Blocks. Each Directory is initialized to holds 2 Directory Entries with human-readable names:
"." : corresponds to the inode of the Directory itself
".." : corresponds to the inode of the Directory's parent Directory

iii) The Filesystem Root inode is a Directory. Since it has no parent Directory, the Directory Entry named ".." is also associated with inode number 2 (the Filesystem Root itself).

You are given an implementation of a simplified version of the ext2 Filesystem. This implementation **simulates** the existence of a Disk by allocating Virtual Memory space. It also considers a Filesystem with a **single Block Group**.
This implementation also provides 1) a function to initialize the Filesystem, 2) one to "dump" it, i.e. perform a depth-first-search crawl from the Filesystem Root inode and print out all inodes and their information, and 3) one to "crawl" it, i.e. to perform the same procedure but only print out the Filesystem tree, i.e. only the human-readable names of files and directories.

Explanations about the provided data structures and code:

```
#define BLKSIZE 4048
```

It is assumed that the fixed-size of a single block is 4KB

```
const int root_inode_number = 2;
const int badsectors_inode_number = 1;
const int invalid_inode_number = 0;

const int root_datablock_number = 0;
```

As mentioned above for inode numbers 0, 1, 2;
For simplicity this implementation does not initialize or use inode number 1 (the bad data block tracking one) and therefore we can assume that the first Data Block (number 0) can be used by the root inode number 2 to store its Directory Entries.

```
typedef struct _block_t {
  char data[BLKSIZE];
} block_t;
```

A DataBlock holds **`BLKSIZE`** number of bytes.

```
typedef struct _inode_t {
  int size;  // of file in bytes
  int blocks;  // blocks allocated to this file - used or not
  block_t* data[15];  // 12 direct, 3 indirect
} inode_t;
```

An inode holds the following information:
`size`: The size in Bytes. If the inode corresponds to regular file, it is the size of its data. If the inode corresponds to a directory, then its data will only contain Directory Entries (**`struct dirent_t`**) and therefore the size will be: # of Directory Entries * **`sizeof(dirent_t)`** for the simplified implementation treated in this assignment, where in **`dirent_t`** the **`name`** field is a fixed-size character array (see below).
`blocks`: The number of Blocks allocated to this file (whether used or not).
`data`: An array of **`block_t`** Pointers, each holding the address of a Block on the Filesystem. The first 12 are "Direct Blocks", i.e. the data in the Block correspond to actual data of the file/directory. The next one (13th) is an "Indirect Block", i.e. the data in the Block correspond to **`block_t`** Pointers (**`BLKSIZE/ADDRSIZE`** in total) with addresses of other Blocks, that they themselves contain the actual data of the file/directory. Then the next one (14th) is a "Double-Indirect Block" which adds one more level of indirection, and the next one (15th) is a "Triple-Indirect Block" which adds one more.

Note: For this simplified implementation you will not be required to use more than 1 Block for each inode, i.e. only **`data[0]`** will be used to store file/directory data, but you will have to remember to initialize all the rest of the **`block_t`** Pointers in the **`data`** array to NULL.

```c
typedef struct _dirent_t {
  int inode;  // entry's inode number
  char file_type;  // entry's file type (0: unknown, 1: file, 2: directory)
  int name_len;  // entry's name length ('\0'-excluded)
  char name[255];  // entry's name ('\0'-terminated)
} dirent_t;
```

A Direntory Entry holds the following information:
inode: The inode number (index of the inode in the inode Table).
file_type: The type of the file that this inode describes (file/directory). Remember, the **struct inode_t** itself does not store that information, it is instead encoded within the **dirent_t** that references the specific inode.
name_len: Length of the human-readable name of the Directory Entry (filename/directory name). Excludes the NULL-terminating character (like **strlen()**).
name: Character array (fixed-size) to hold the human-readable name, which has to be NULL-terminated.

Note: In the ext2 Filesystem, the **name** is variable-length. This allows flexibility, and usually more Directory Entries (assuming short names) to fit inside a single Block. In this simplified representation of this assignment however, the fact that it is fixed-size allows to easily be able to figure out how many Directory Entries exist within a directory inode.
I.e. given an inode that you know to be a directory type one, you can do:
# of Directory Entries of **inodeX** : **inodeX.size * sizeof(dirent_t)**

```c
typedef struct _superblock_info_t {
  int blocks;  // total number of filesystem data blocks
  char name[255];  // name identifier of filesystem
} superblock_info_t;

union superblock_t {
  block_t block;  // ensures superblock_t size matches block_t size (1 block)
  superblock_info_t superblock_info;  // to access the superblock info
};
```

A Superblock contains information about the entire Filesystem and its structure. Here we are using **superblock_info_t** to only store a **name** for the Filesystem (like the name you give to your Flash Drive when you format it) and the total number of **blocks** in the Filesystem.
One important thing to note is that, as described in the beginning, the Superblock takes up exactly one Block. A nice way to make sure that the **superblock_info_t** will take up an entire Block is to use a **union**. Using this, we get the final definition of our **superblock_t**. Assuming we have a **superblock_t**-type variable named **super** in our hands, we can access its **superblock_info_t**-type contents (e.g. the **name**) by using **union** syntax, i.e. doing: **super.superblock_info.name**

```c
typedef struct _groupdescriptor_info_t {
  inode_t* inode_table;  // location of inode_table (first inode_t in region)
  block_t* block_data;  // location of block_data (first block_t in region)
} groupdescriptor_info_t;

union groupdescriptor_t {
  block_t block;  // ensures groupdescriptor_t size matches block_t size (1 block)
  groupdescriptor_info_t groupdescriptor_info;  // to access the groupdescriptor info
};
```

A Group Descriptor contains information about the Disk location of information of the structure of

a Group, for example:
inode_table: The starting location of the array inode Table (**inode_t** Pointer)
block_data: The starting location of the array of Data Blocks of the Filesystem (**block_t** Pointer)
As mentioned, the ext2 Filesystem has multiple Groups, and their Group Descriptors are placed inside the Group Descriptor Table, which can take up multiple Blocks of the Filesystem.

For this simplified implementation where we deal with a single Group, we assume we have no Group Descriptor Table and this just one Group to worry about. Similarly to before we make sure our **groupdescriptor_info_t** is placed within a **union** called **groupdescriptor_t**, in order to make sure that **groupdescriptor_t** takes up the size of 1 Block. Assuming we have a **groupdescriptor_t**-type variable named **group** in our hands, we can access its **inode_table** by using **union** syntax, i.e. doing: **group.groupdescriptor_info.inode_table**

IMPORTANT Note: The reason why we want to have the Superblock and the Group Desriptor take up integral multiples of 1 Block is Alignment. We do not want different entities of the Filesystem to end up spanning different Blocks (and therefore have to read/write 2 Blocks to access the first half and then the other half of the data we need). For the same reasons, the Block Bitmap, the inode Bitmap, and the inode Table on the Disk are also integral 1-Block multiple-long ,and of course the Data Blocks area on the Disk comprises of an integral Block-multiple as well.

```
typedef struct _myfs_t {
  union superblock_t super;  // superblock
  union groupdescriptor_t groupdescriptor;  // groupdescriptor
  block_t bmap;  // (free/used) block bitmap
  block_t imap;  // (free/used) inode bitmap
} myfs_t;
```

The Block-aligned Filesystem structure is therefore achieved by the struct **myfs_t**:
super: A Superblock at the beginning of the Filesystem (beginning of **myfs_t**) which is 1 Block-long (its type is **union superblock_t**).
groupdescriptor: A Group Descriptor at the immediately next spot of the Filesystem (next field of **myfs_t**) which is also 1 Block-long (its type is also **union superblock_t**).
bmap: Block Bitmap that is 1 Block-long (its type is **block_t**). This Bitmap stores bitwise the state (0:free / 1:occupied) of the Data Blocks of the Group.
imap: inode Bitmap that is 1 Block-long (its type is **block_t**). This Bitmap stores bitwise the state (0:unused / 1:used) of the inodes of the Group (which reside within the inode Table).

```
myfs_t* my_mkfs(int size, int maxfiles);
```

Function to initialize the Filesystem. **size** is the total size of the Filesystem Data Blocks area, and **maxfiles** is the max number of files/directories it will support (remember, the inode Table is fixed-size and allocated at the creation of the Filesystem).

Some notes about what is going on inside the function:

```
int num_data_blocks = roundup(size, BLKSIZE);
int num_inode_table_blocks = roundup(maxfiles*sizeof(inode_t), BLKSIZE);
size_t fs_size = sizeof(myfs_t) +
                 num_inode_table_blocks * sizeof(block_t) +
                 num_data_blocks * sizeof(block_t);
```

Calculation of how much space needs to be reserved for the entire Filesystem. It is required to calculate how many Data Blocks we need **roundup(size/BLKSIZE)**, how many Blocks the inode Table will require **roundup(maxfiles*sizeof(inode_t/BLKSIZE)** (simplified, inode should actually not be split between Blocks), and finally sum it all up (Superblock + Group Descriptor + Block Bitmap + inode Bitmap + inode Table + Data Blocks). We then perform

memory allocation (**malloc**) to get back Virtual Memory which is how we will be simulating the Disk storage in this assignment.

```
myfs_t* myfs = (myfs_t*)ptr;
```
We reinterpret (Pointer-cast) the Virtual Memory we got back from **malloc** as a **myfs_t** type to start working on the Filesystem data and set it up.

```
// superblock
void *super_ptr = calloc(BLKSIZE, sizeof(char));
// read-in (not required, we are creating filesystem for first time)
union superblock_t* super = (union superblock_t*)super_ptr;
super->superblock_info.blocks = num_data_blocks;
strcpy(super->superblock_info.name, "MYFS");
// write out to fs
memcpy((void*)&myfs->super, super_ptr, BLKSIZE);
```
First we update the information inside the Superblock. To be able to do so, we need to simulate the same thing that would happen with an actual Disk, which means we Read-Modify-Write data! Since we are just initializing the Filesystem, reading-in the previous information of the Superblock is not applicable. We allocate some new Memory (zero-initialized using **calloc**) that is as long as a Block, then reinterpret it as a **superblock_t**, modify its fields, and finally write it out to the Disk (since we are simulating Disk operations using Virtual Memory, we **memcpy** to the Address of **myfs->super** an amount of Bytes equal to 1 Block size (**BLKSIZE**) since this it the atomic unit of reading and writing for a Disk).

```
// groupdescriptor
void *groupdescriptor_ptr = calloc(BLKSIZE, sizeof(char));
// read-in (not required, we are creating filesystem for first time)
union groupdescriptor_t* groupdescriptor =
  (union groupdescriptor_t*)groupdescriptor_ptr;
groupdescriptor->groupdescriptor_info.inode_table =
  (inode_t*)((char*)ptr +
             sizeof(myfs_t));
groupdescriptor->groupdescriptor_info.block_data =
  (block_t*)((char*)ptr +
             sizeof(myfs_t) +
             num_inode_table_blocks * sizeof(block_t));
// write out to fs
memcpy((void*)&myfs->groupdescriptor, groupdescriptor_ptr, BLKSIZE);
```
Then we update the information inside the Group Descriptor. We have to again simulate the Read-Modify-Write paradigm of writing to Disk, which works similarly as above. As far as the important Superblock information goes, we compute the Address of the **inode_table** inside the Filesystem (address of the start of the Filesystem (**ptr**) offseted by the Superblock, Group Descriptor, Block Bitmap, and inode Bitmap, which is effectively the **sizeof(myfs_t)**). Then we compute the Address of the **block_data** inside the Filesystem (previous offset + the size of the entire inode Table).

Note: For simplicity, the numbers fed to **my_mkfs()** of this example ensure that only 1 Block will be necessary for the inode Table.

Then we proceed to crate the Filesystem Root:

```
// inode
void *inodetable_ptr = calloc(BLKSIZE, sizeof(char));
// read-in (not required, we are creating filesystem for first time)
inode_t* inodetable = (inode_t*)inodetable_ptr;
inodetable[root_inode_number].size = 2 * sizeof(dirent_t);
inodetable[root_inode_number].blocks = 1;
for (uint i=1; i<15; ++i)
  inodetable[root_inode_number].data[i] = NULL;
inodetable[root_inode_number].data[0] =
  &(groupdescriptor->groupdescriptor_info.block_data[root_datablock_number]);
// write out to fs
memcpy((void*)groupdescriptor->groupdescriptor_info.inode_table,
  inodetable_ptr, BLKSIZE);
```

Here we set up the inode that represents the Filesystem Root. Again we simulate the Read-Modify-Write paradigm.
Therefore there is no notion of accessing a single inode in the inode Table in isolation; we get, modify, and write the entire inode Table, and if we want to modify a single inode we need to index the Table at the corresponding inode number location (**`inodetable[root_inode_number]`**).
As far as the inode information goes:
`size`: Set to indicate that 2 Directory Entries (for '.' and '..') are stored
`blocks`: Set to indicate that this inode only has 1 Data Block allocated to it (for 2 Directory Entries it's enough at initialization, and for simplicity this assignment won't need to expand to more Data Blocks).
`data[]`: Mappings of Data Blocks 2-15 (indexes 1-14) are initialized to NULL to indicate that no valid Data Block is mapped. For the first Data Block **`data[0]`**, we map it to the Address of a Free Data Block in the Filesystem (Address-of **`block_data[root_datablock_number]`**). Since the Filesystem is just being initialized, we can assume that the first Data Block will be available (**`root_datablock_number=0`**) so we don't have to search the Block Bitmap for a Free Block.

```
// data (directory)
void *dir_ptr = calloc(BLKSIZE, sizeof(char));
// read-in (not required, we are creating filesystem for first time)
dirent_t* dir = (dirent_t*)dir_ptr;
// dirent '.'
dirent_t* root_dirent_self = &dir[0];
root_dirent_self->name_len = 1;
root_dirent_self->inode = root_inode_number;
root_dirent_self->file_type = 2;
strcpy(root_dirent_self->name, ".");
// dirent '..'
dirent_t* root_dirent_parent = &dir[1];
root_dirent_parent->name_len = 2;
root_dirent_parent->inode = root_inode_number;
root_dirent_parent->file_type = 2;
strcpy(root_dirent_parent->name, "..");
// write out to fs
memcpy((void*)(inodetable[root_inode_number].data[0]), dir_ptr, BLKSIZE);
```

We proceed to create the data of the Root inode, which will be a list of Directory Entries. To do so, we again use the Read-Modify-Write paradigm. Since we are just now initializing the Data, we don't have read it into Memory from the corresponding Data Block on the Disk, we can instead allocate zero-initialized Memory (**`calloc`**) and reinterpret it (Pointer-cast) as an array of Directory Entries (**`dirent_t*`**).

We now index into the first (index-0) element of this array to create the Directory Entry for '.' by filling up the **name** (and **name_len**) and the **inode** number (which is the inode itself). We then index into the second (index-1) element of this array to create the Directory Entry for '..' similarly. *Note:* The case of the Root Directory is special as it has no Parent inode, and therefore the inode number for '..' is also set to the inode itself. Finally we write out the entire Block of data into the appropriate location of the Filesystem (the Disk location pointed-to by the 0th Data Block of the inode at index 2 (Root inode) inside the inode Table).

```
// data  bitmap
void* bmap_ptr = calloc(BLKSIZE, sizeof(char));
// read-in (not required, we are creating filesystem for first time)
block_t* bmap = (block_t*)bmap_ptr;
bmap->data[root_datablock_number / 8] |= 0x1<<(root_datablock_number % 8);
// write out to fs
memcpy((void*)&myfs->bmap, bmap_ptr, BLKSIZE);
```

We then need to populate the Block Bitmap. Again with the Read-Modify-Write paradigm, but since we are just initializing the Filesystem we just create a zero-initialized Block which we will modify and then write-out to the Filesystem at the end. In order to be able to set a particular bit of an entire Block we need a clever way:
block_t.data is an array of Bytes (8-bits long each). For a particular bit number (e.g. the 12th bit) inside the Block, to find the index of that array that corresponds to that bit we can do integer divion by the fundamental size of an element of the array (8). For example to access the 12th bit we need to access the 2nd Byte (index-1 element of the array), i.e. **12/8**. Then to access the appropriate bit location within that we just found Byte, we can use the modulo result by the fundamental size of an element (8). For example the 12th bit is the 4th bit, i.e. **12%8**, of the 2nd Byte we found in the previous step. So overall, we can use left-shifting and bitwise OR-in like so:

```
bitmap[i / 8]  |= 0x1 << (i % 8);
```

to set the **i**-th bit of a Bitmap array named **bitmap**.
Here we set the 0-th (**root_datablock_number**) bit of the Block Bitmap to indicate that this Block is now reserved (used by the Root inode to store its Directory Entries).

```
// inode  bitmap
void* imap_ptr = calloc(BLKSIZE, sizeof(char));
// read-in (not required, we are creating filesystem for first time)
block_t* imap = (block_t*)imap_ptr;
imap->data[root_inode_number / 8] |= 0x1<<(root_inode_number % 8);
imap->data[badsectors_inode_number / 8] |= 0x1<<(badsectors_inode_number % 8);
imap->data[invalid_inode_number / 8] |= 0x1<<(invalid_inode_number % 8);
// write out to fs
memcpy((void*)&myfs->imap, imap_ptr, BLKSIZE);
```

Finally we need to populate the inode Bmap. We employ the same strategy. The key difference is that we need to indicate that not just the 2nd inode (**root_inode_number**) is in use, but also the other reserved numbers as well (**badsectors_inode_number**, **invalid_inode_number**) to make sure that in any future attempt to create a new inode (to create a new file/directory), these will not be found as unused.

You are also provided with the functions:

```
void my_dumpfs(myfs_t* myfs);
```

and:

```
void my_crawlfs(myfs_t* myfs);
```

which allow you to print out the state of the Filesystem (for debugging).

Your task is to write a function that creates a new Directory in the Filesystem:

```
void my_creatdir(myfs_t* myfs,
                 int cur_dir_inode_number,
                 const char* new_dirname);
```

1st param: **myfs_t***, Pointer to a Filesystem that was created with **my_mkfs()**.
2nd param: **int**, the inode number of the Directory that will be the Parent Directory of the newly created one
3rd param: C-string, the name of the newly created Directory.

Your function should:

1) Access the inode Bitmap of the Filesystem and search to find the first location that can be used for the new inode you will create (look above about how to access a specific bit inside a Bitmap array). Then set that bit to indicate this inode is now used, and update the Filesystem.

   CAUTION: You are not allowed to directly access information on the Filesystem, such as: **myfs->imap.data[…]**. You have to use the Read-Modify-Write paradigm as an actual Filesystem-implementing code would do, i.e. you have to:
   i) **malloc** space in Memory to read-in from Disk (you have to malloc 1 Block) to read 1 Block from Disk.
   ii) read-in from Disk (use **memcpy** to simulate this using Virtual Memory, i.e. copy from **myfs->imap** to the space you **malloc**'ed in step i).
   iii) work on the data in Memory, i.e. find the first unused (0) bit, and set it as used (1).
   iv) write-out to Disk (use **memcpy** to simulate this using Virtual Memory, i.e. copy from the space you **malloc**'ed in step i) to the **myfs->imap**).

2) Use Read-Modify-Write again. Access the Block Bitmap of the Filesystem and search to find the first location that can be used as a Data Block to store the data of the new inode you are creating. Then set that bit to indicate this Data Block is now used, and update the Filesystem.

3) Use Read-Modify-Write again. Access the inode Table of the Filesystem and load the Parent Directory inode, and the new Directory inode.
   Modify the Parent Directory inode's **size** to indicate that it will now have 1 more Directory Entry.
   Initialize the new Directory inode (**size**, **blocks**, **data**). Use the previously provided code that initialized the Root Directory inode about how to achieve that. *Note:* Your new Directory is expected to hold 2 Directory Entries at initialization, for '.' and '..'.
   Then finally update the inode Table on the Filesystem.

4) Use Read-Modify-Write again. Access the Parent Directory's data from the Filesystem. As mentioned, the data will be inside the Data Block location pointed to by the **inode.data[0]** since this example uses no more that 1 Block.
   Modify the Parent Directory's data to append the new Directory Entry you are creating. Follow the example above that does indexing (**&dir[0]**, **&dir[1]**) after reinterpreting (Pointer-casting) the Block memory to a **direntry_t** Pointer (i.e. to access the Memory like it has the layout of a **direntry_t** array). Your code has to be generic, i.e. has to be able to find out how many Directory Entries were in there before, and modify the next one. The new Directory Entry's **inode** will be the inode number that you are currently creating, and the **name** will be the one passed to the function. You are allowed to use **strlen()** for the **name_len**.
   Then finally update the Parent Directory's data on the Filesystem.

5) Use Read-Modify-Write again. Access the new Directory's data from the Filesystem (the

Data Block location you mapped to the new inode's **`inode.data[0]`** in step 4). Load it into Memory and modify it to include 2 Directory Entries for '.' and '..'. The example provided above for the Root Filesystem does that exactly.
Finally update the new Directory's Data Block on the Filesystem.

You are given a sample **`main()`** that first creates a Filesystem using **`my_mkfs()`** (already implemented) then creates some Directories within it using **`my_creatdir()`** (the one you need to implement yourselves), and finally prints out the Filesystem structure using **`my_dumpfs()`, `my_crawlfs()`** (already implemented as well).

```
int main(int argc, char *argv[]){

  inode_t* cur_dir_inode = NULL;

  myfs_t* myfs = my_mkfs(100*BLKSIZE, 10);

  // create 2 dirs inside [/] (root dir)
  int cur_dir_inode_number = 2;  // root inode
  my_creatdir(myfs, cur_dir_inode_number, "mystuff");  // will be inode 3
  my_creatdir(myfs, cur_dir_inode_number, "homework");  // will be inode 4

  // create 1 dir inside [/homework] dir
  cur_dir_inode_number = 4;
  my_creatdir(myfs, cur_dir_inode_number, "assignment5");  // will be inode 5

  // create 1 dir inside [/homework/assignment5] dir
  cur_dir_inode_number = 5;
  my_creatdir(myfs, cur_dir_inode_number, "mycode");  // will be inode 6

  // create 1 dir inside [/homework/mystuff] dir
  cur_dir_inode_number = 3;
  my_creatdir(myfs, cur_dir_inode_number, "mydata");  // will be inode 7

  printf("\nDumping filesystem structure:\n");
  my_dumpfs(myfs);

  printf("\nCrawling filesystem structure:\n");
  my_crawlfs(myfs);

  return 0;
}
```

Your code has to work such that you get the following Directory structure output:

```
Crawling filesystem structure:
/
|_ .
|_ ..
|_ mystuff
   |___ .
   |___ ..
   |___ mydata
      |_____ .
      |_____ ..
|_ homework
   |___ .
   |___ ..
   |___ assignment5
      |_____ .
      |_____ ..
      |_____ mycode
         |_______ .
         |_______ ..
```

General Directions:
Fill in the provided source code file **`myfs.c`** to complete the implementation of **`my_creatdir()`**. This is the C code you are expected to turn in.

Submission Directions:
When you are done, create a directory named **`PA5_Lastname_Firstname`**, place your **`Makefile`** (which has to have at least one default target that builds your **`mymalloc`** application executable), and your **`myfs.c`** source code file into it, and then compress it into a **`.zip`** or **`.tar.gz`** with the same filename as the folder.
Upload the archive (compressed) file on Webcampus.

Early/Late Submission: You can submit as many times as you would like between now and the due date.
A project submission is "late" if any of the submitted files are time-stamped after the due date and time. Projects that are up-to 24 hours late will receive a 15% penalty, ones that are up-to 48 hours late will receive a 35% penalty, ones that are up-to 72 hours late will receive a 60% penalty, and anything turned in 72 hrs after the due date will receive a 0.

Verify your Work:
You will be using **`gcc`** to compile your code. Use the **`-Wall`** switch to ensure that all warnings are displayed. Strive to minimize / completely remove any compiler warnings; even if you don't warnings will be a valuable indicator to start looking for sources of observed (or hidden/possible) runtime errors, which might also occur during grading.

After you upload your archive file, re-download it from WebCampus. Extract it, build it (e.g. run **`make`**) and verify that it compiles and runs on the ECC / WPEB systems.

- Code that does not compile will receive an automatic 0.