Appendix

Some implementation needs for A#4

Byte ordering

Some content of superblock and directory entries are *integers* (either two-byte for four-byte). These integers are to be written to disk images along with other file-system metadata. However, the order in which an integer's bytes are written to a file depends upon the *endianness* of the host computer. Consider this short program (line numbers are for reference):

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     short ii = 127;
6     int   i  = 127;
7
8     FILE *f = fopen("somefile.bin", "w");
9     if (f == NULL) { exit(1); }
10     fwrite(&ii, sizeof(short), 1, f);
11     fwrite(&i, sizeof(int), 1, f);
12     fclose(f);
13 }
```

This is compiled and executed on *linux.csc.uvic.ca* with the contents of *somefile.bin* displayed on the console.

```
$ gcc endian.c -o endian
$ ./endian
xxd somefile.bin
0000000: 7f00 7f00 0000                            ......
$
```

Because of the *little-endian* architecture of *linux.csc.uvic.ca*, bytes in a *short* (two-byte integers) and bytes in an integer (four-bytes long, also known as *long*) are in reverse order (as seen in the output above produced by *xxd*).

However, we are used to reading bytes from left to right, not right to left (i.e., *big-endian*). Integers in file-system metadata must be stored big-endian order as this will greatly help during debugging. In order to ensure this, we can use code in the network library, i.e., data sent over the network is expected to be in big-endian format. If we add *#include <arpa/inet.h>* to the list of includes, and change lines 5 and 6 as follows:

```
    short ii = htons(127);
    int   i  = htonl(127);
```

then after recompiling and re-running the program, we examine the contents of *somefile.bin*:

```
$ xxd somefile.bin
0000000: 007f 0000 007f                          ......
```

and see that the bytes are now ordered as we expect for a short and a long integer.

Please become familiar with the functions *htons()*, *htonl()*, *ntohs()* and *ntohl()*. You should use these to convert data from and to the disk image as needed (i.e., data should be in host order for calculations, but network order for storage). All disk images provided for you in the assignment have been stored in big-endian order.

Packed structs

One of the files provided to you is *disk.h*. Amongst other things in this header file – such as a large number of *#define* directives – there are declared two types: *superblock_entry_t* and *directory_entry_t*. These can be used to read superblocks and directory entries from the disk image.

At the end of each type declaration, however, is a strange compiler directive. It is needed in order to defeat the compiler's normal layout of fields within a struct. Normally the compiler ensures fields are always aligned on a word boundary (i.e., one word = four bytes). There are good reasons for this, having to do with the way hardware retrieves data from RAM. For our disk image, though, such alignment causes problems as we want fields to appear precisely where we put them in the structure. In order to ensure this precise layout we use the *packed* directive. Now we can directly read and write instances of *superblock_entry_t* and *directory_entry_t* using C file routines.

File routines

You will make heavy use of the library functions *fopen*, *fseek*, *fread* and *fwrite*. There are lots of resources online describing these functions and you are encouraged to read them. There will be some discussion of these during the tutorial for this assignment.