Objectives

- Project #3:
  Loading and Executing Programs + Shell

Our File System

- **Disk Directory**: keeps track of
  - The names of files that are stored on the disk
  - The sectors that make up each file.
- **Disk Map**: keeps track of which sectors on the disk are used and which are free.

You can look at loadfile.c to see how these are updated when a file is added to the file system

Disk Map

- 512 one-byte entries
- Entry *i* holds:
  - 0x00 if i^{th} sector is free
  - 0xFF if i^{th} sector is used
- Disk map will be stored in absolute sector 1 of the disk
  - right after the bootloader

Disk Directory

- Holds 16 32-byte entries
  - 6 character file name (0x00 padded)
    - Not necessarily null terminated
  - 26 bytes – each indicates a sector (0x00 padded)
  - Example:

Name padding

```
46 49 4C 45 00 00 10 11 A0 39 52 00 00 00 00 00 00 00...
F  I  L  E
```

- Disk Directory will be stored in sector 2 of the disk
  - (right after the Map and before the kernel)

Disk Directory Visualization

floppya.img

0	Bootloader
1	Disk Map
2	Disk Dir
3	KERNEL
…	…
19	message.txt
20	Bigfile
21	Bigfile
22	Bigfile
23	FILE
…	…
…	…

Disk Directory ←how big is this?

32 characters "wide"

K	E	R	N	E	L	3	4	5	6	…
m	e	s	s	a	g	19	0	0	0	…
B	i	g	f	i	l	20	21	22	0	…
F	I	L	E	0	0	23	0	0	0	…
…										

16 entries

You can see what your disk directory looks like by using `hexdump -C floppya.img`

Review: Pointer Arithmetic

```
main() {
   char str[52];
   fill(str);
   str[51] = '\0';
   printf("%s\n\r", str);
}

void fill(char *buf) {
   int i;
   for (i=0; i<26; i++) {
      buf[0] = 'a' + i;
      buf[1] = '-';
      buf = buf + 2;
   }
}
```

What does this program display?

Review: Pointer Arithmetic

```
main() {
   char str[52];
   fill(str);
   str[51] = '\0';
   printf("%s\n\r", str);
}

void fill(char *buf) {
   int i;
   for (i=0; i<26; i++) {
      buf[0] = 'a' + i;
      buf[1] = '-';
      buf = buf + 2;
   }
}
```

- The value of a pointer can be changed using standard arithmetic operators.
  - E.g. Adding 2 increases value of pointer by 2*(size of the stored data).

Disk Directory

If `diskDirectory` is a pointer to the start of an entry, how can you find the sectors that file is in?

- Holds 16 32-byte entries
  - 6 character file name (0x00 padded)
    - Not necessarily null terminated
  - 26 bytes – each indicates a sector (0x00 padded)
  - Example:

Name padding

```
46 49 4C 45 00 00 10 11 A0 39 52 00 00 00 00 00 00 00...
F  I  L  E
```

- Disk Directory will be stored in sector 2 of the disk
  - (right after the Map and before the kernel)

C Program Organization

- .h files contain function prototypes
  - Included in any .c file that wants to call the prototyped functions.
- .c files contain implementation
  - Linked with other .c files that call the functions.

printInt.h

```
#ifndef PRINT_INT
#define PRINT_INT
void printIntArr(int *arr, int len);
#endif
```

printInt.c

```
#include "stdio.h"
#include "printInt.h"
void printIntArr(int *arr, int len) {
    int i;
    for (i=0; i<len; i++) {
       printf("%d ", arr[i]);
    }
    printf("\n\r");
}
```

myProg.c

```
#include "stdio.h"
#include "printInt.h"
main() {
    int vals[3];
    vals[0] = 7;
    vals[1] = 9;
    vals[2] = 22;
    printIntArr(vals,3);
}
```


Makefile for Previous Slide's Code

```
myProg: myProg.c printInt.o
     gcc -o myProg myProg.c printInt.o

printInt.o: printInt.h printInt.c
     gcc -c printInt.c printInt.h
```

Reminder: we're using bcc to compile most of our programs in the project

Makefile Example for bcc

```
myProg: myProg.o printInt.o
     ld86 -o myProg -d myProg.o printInt.o

myProg.o: myProg.c
     bcc –ansi –c –o myProg.c

printInt.o: printInt.h printInt.c
     bcc –ansi –c –o printInt.c
```

We need the .h file as a dependency but it is not compiled

Functional Decomposition

- Create functions for logical units of work
- Some useful functions might include:
  - String comparison
    - E.g. for comparing filenames
  - Finding the directory entry for a filename.
  - And others you find useful...
- Test individually – make sure functions work

Review: C Structures

- A ***struct*** is a logical (and physical) grouping of variables.
- Use `name.field` to access the structure's fields.
  - `file.name[3] = 'x';`
  - `file.sectors[0] = 0x22;`

```
struct dirEntry {
   char name[6];
   char sectors[26];
};
```

There is no byte type in C, so we will use chars for bytes

```
main() {
    struct dirEntry file;
    file.name[0] = 'F';
    file.name[1] = '1';
    file.name[2] = 0x00;
    printf("%s\n\r", file.name);
}
```


Example: Nested C Structures

```
struct dirEntry {
   char name[6];
   char sectors[26];
};
struct directory {
   struct dirEntry
entries[16];
};
```

```
main() {
   struct directory diskDir;
   diskDir.entries[3].name[0] =
      'A';
   diskDir.entries[3].name[1] =
      'B';
   diskDir.entries[3].name[2] =
      0x00;
   printf("%s\n\r",
      diskDir.entries[3].name);
}
```

Using a Structure as a Buffer

```
main() {
    struct directory diskDir;
    fill((byte*)&diskDir);
    printf("%s\n\r",
    diskDir.entries[2].name );
}

void fill(byte *buf) {
    buf[64] = 'A';
    buf[65] = 'B';
    buf[66] = 0x00;
}
```

- A structure is a contiguous collection of bytes.
  - Can fill structure with bytes.
  - Can access structure using fields.

Passing Structure to a Function

```
void useStruct(struct directory *dir);

main() {
    struct directory diskDir;
    // here we have a directory structure...
    // use . to access fields
    diskDir.entries[2].name[0] = 'A';
    diskDir.entries[2].name[1] = 'B';
    diskDir.entries[2].name[3] = 0x00;
    readSector((char *)&diskDir, 2);
    useStruct(&diskDir);
    printf("%s\n\r", diskDir.entries[2].name);
}
void useStruct(struct directory *dir) {
    // here we have a pointer to a directory structure
    // use -> to access fields
    dir->entries[2].name[0] = 'A';
    dir->entries[2].name[1] = 'B';
    dir->entries[2].name[3] = 0x00;
}
```

We need to pass around the struct as a pointer instead of by copy; otherwise, we get an error about memcpy when trying to compile