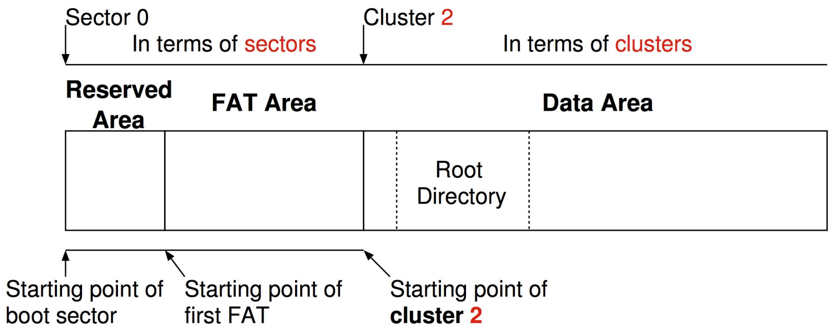
Abusing the System

By: Stuart Nevans Locke, Jacob Doll, Eric Chen

Overview of the Project and Major Components

- End goal was writing a shell-like program that could take files (specifically binary files), even ones compiled from other sources, and let us run them on the OS
- We worked on three projects to attempt this:
  - File System
  - Paging and Virtual Memory
  - ELF Loader

File System Design



- FAT32 File System
- Made up of three core areas:
  - The Boot Record (Reserved Area)
    - BIOS Parameter Block (BPB) & Extended Boot Record
    - Usually located in logical sector 0
    - Used for loading the operating system into memory, but also contains data about file system
  - The File Allocation Table (FAT)
    - “Table of Contents of Disk”
    - Keeps track of status and location of all clusters in the disk (i.e. which clusters are being used by what file and directory)
  - The Directory and Data area
    - Contains the actual files and data
    - Directory entries store about where a file’s data is and information about the file itself (Root Directory), while the data is stored in clusters in the rest of the Data Area

File System Implementation and Design (Setting up FAT32)

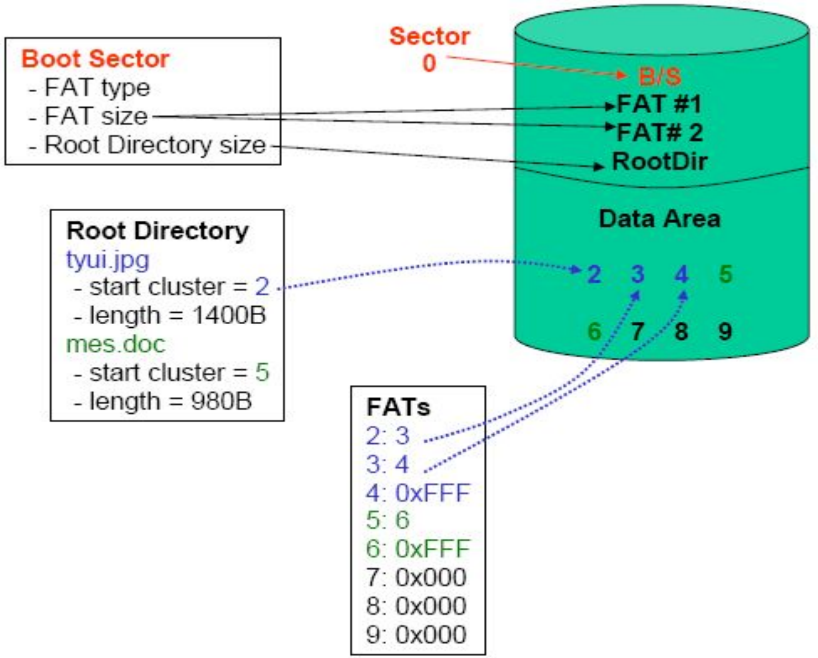
- First get information about file system and disk from the boot record and store it in a BIOS Parameter Block
- Information is gotten from sector 0 of the disk using an ATA port driver
- Information from BPB is used to find the beginning of the FAT and Data Area sectors and set up the FAT and its sectors which is stored in a file system structure for later usage

```c
/*
** BIOS Parameter Block that contains information about the boot record
**
*/
typedef struct bios_param_block{
    char jmp[3];
    char oem[8];
    uint16_t bytes_per_sector;
    uint8_t sectors_per_cluster;
    uint16_t reserved_sectors;
    uint8_t num_FAT;
    uint16_t num_root_dir;
    uint16_t total_sectors; //If 0 then > 65535 so use large_sector_count
    uint8_t media_descriptor_type;
    uint16_t num_sectors_per_FAT; //Only used if this was FAT12/FAT16
    uint16_t num_sectors_per_track;
    uint16_t num_heads_media;
    uint32_t num_hidden_sectors;
    uint32_t large_sector_count; //Used if more than 65535 sectors in the volume

  //Extended Boot Record
  uint32_t sectors_per_FAT32;
  uint16_t flags;
  uint16_t FAT_version_num;
  uint32_t root_dir_cluster_num;
  uint16_t sector_num_FSInfo;
  uint16_t sector_num_backup;
  char reserved_0[12];
  uint8_t drive_num;
  uint8_t windows_flags; //Only used for flags in Windows NT, reserved otherwise
  uint8_t signature;
  uint32_t volume_id;
  char volume_label[11];
  char system_id[8];
} bpb_t;
```

File System Implementation (Reading Directories)

1. Find the root directory cluster (for FAT32 it's in the extended Boot Record (it's often 2))
2. Use it to find the first cluster number for the directory entry and then calculate the first sector of that cluster
   a. first_sector_of_cluster = ((cluster - 2) * fat_boot->sectors_per_cluster) + first_data_sector);
3. Read the entry from the sector (if the first byte is 0 then there are no more files/directories to be read and we're done) (if the first byte is 0xE5 then this specific entry is unused and we move onto the next entry) (Each entry is 32 bytes in a 512 byte sector, max 16 entries per sector)
4. Once all entries have been read from the cluster we check if there is another cluster in the cluster chain by checking FAT[current_cluster] == 0x0FFFFFF8
5. If true, then end of cluster chain found, else go to next cluster found from FAT[current_cluster] and repeat starting from step 2




```
/*
** FAT Filesystem that contains the boot record, the file allocation table (FAT)
** and information about the sectors, clusters, and where they begin.
** Similar to FSInfo Structure
*/
typedef struct FAT32_struct {
    bpb_t bios_block;
    uint32_t *FAT;
    uint32_t data_begin_sector;
    uint32_t FAT_begin_sector;
    uint32_t current_cluster_pos;
} f32_t;
```

File System Problems Encountered

- Cluster vs Sector:
  - Difficulty keeping track of them and their sizes
  - Sectors - usually 512 bytes, Clusters - groups of sectors
- Differences between FAT12, FAT16, and FAT32
  - Each one has different implementations and different extended boot sectors
  - Many sources online cover all three at once, so it is somewhat difficult to find the necessary information for only FAT32

File System Future Things

- Add support for Long File Names
  - Current file system lacks the ability to add long file names to the directory, only allows the standard 8.3 directory entry
- Add an actual FSInfo Structure
  - Current file system has a structure that has the important information that would be found from a FSInfo Structure, but an actual one would make the current file system more like FAT32
- Add additional functionality to the file structure
  - Current file system lacks support for searching through and reading subdirectories

Paging and Virtual Memory

- Design
  - Multilevel page table
  - Page Directory Entry
  - Page Table Entry
- Page directory goes in cr3, CPU flushes the TLB and handles the rest
- Unique Address Space Per Process
  - Add an entry to the pcb for each page directory
  - We can have multiple things at the same address which is nice

Virtual Memory Implementation

- Important to do very early on during boot
  - Difficult to later change addresses from physical to virtual
- Done as kmem.c is initializing
  - This means we don't have access to the functions from kmem
  - Made another API for physical allocation that the paging code used
- Function to clone page directory
  - Used for fork
- Put stacks around 0xdf000000
- Identity map bottom 600 KB, allocator memory
- Mirror kernel to 0xc0000000

```
pwndbg> monitor info mem
0000000000000000-000000007fe0000 000000007fe0000 -rw
0000000012345000-0000000012346000 0000000000001000 -rw
0000000012347000-0000000012348000 0000000000001000 -rw
0000000012370000-0000000012371000 0000000000001000 -rw
00000000c0000000-00000000c0400000 0000000000400000 -rw
00000000df067000-00000000df077000 0000000000010000 -rw
00000000df1f2000-00000000df1fa000 0000000000008000 -rw
```

Problems Encountered

- Initially tried to statically allocate space for a bunch of page tables in the kernel
    - Resulted in the page table overlapping with graphics memory

- Debugging could be fairly difficult - not immediately obvious where crashes happen
    - Adding a page fault handler made it a bit easier
- Writable bit doesn't seem to matter
    - Seems like you need to enable Write Protection bit in cr0

Paging Future Things

- Create better page fault handler
  - More debug information
  - CoW Pages
- Map kernel at higher address
  - A big chunk of the address space towards the bottom is mapped for the kernel.
  - Would be nice to throw the kernel in the upper 1GB
- Keep better track of pages
  - Don’t identity map page table
  - Better keeping track of what pages to free when process exits
  - CoW Pages
- Add userspace API

Program Loader

- Uses the ELF format
  - Simple format that allows us to load program into memory
- Allows us to provide further separation of kernel/userspace

ELF Format

- Every ELF binary begins with a file header
    - Is found at the beginning of the file
- Binary can be identified using first four bytes
- Contains both section headers and program headers
    - For our purposes only program headers are needed

```
typedef struct {
    uint8_t     e_ident[EI_NIDENT];
    Elf32_Half  e_type;
    Elf32_Half  e_machine;
    Elf32_Word  e_version;
    Elf32_Addr  e_entry;
    Elf32_Off   e_phoff;
    Elf32_Off   e_shoff;
    Elf32_Word  e_flags;
    Elf32_Half  e_ehsize;
    Elf32_Half  e_phentsize;
    Elf32_Half  e_phnum;
    Elf32_Half  e_shentsize;
    Elf32_Half  e_shnum;
    Elf32_Half  e_shstrndx;
} Elf32_Ehdr;
```

```
# define ELFMAG0    0x7F // e_ident[EI_MAG0]
# define ELFMAG1    'E'  // e_ident[EI_MAG1]
# define ELFMAG2    'L'  // e_ident[EI_MAG2]
# define ELFMAG3    'F'  // e_ident[EI_MAG3]
```

Before We Can Begin

- Have to compile user programs outside of kernel
- Each program must have the symbol “_start” to be considered executable by GCC
- Each program has the user lib statically compiled
- Each program stored in physical memory


```
.global  _start
.type  _start, @function
_start:
    jmp main
```

```
$(BUILD_DIR)/usb.img: offsets.h bootstrap.b prog.b prog.nl BuildImage prog.dis user
    ./BuildImage -d usb -o $(BUILD_DIR)/usb.img -b $(BUILD_DIR)/bootstrap.b \
    $(BUILD_DIR)/prog.b 0x10000 \
    $(BUILD_DIR)/sysroot/idle.elf 0x23000 \
    $(BUILD_DIR)/sysroot/main1.elf 0x27000 \
```

ELF Loading Process

1. Verify that address contains valid header
2. Get Program header location from file header
3. Parse Program header
    a. For each program section map physical address to virtual address

Program Header

- Defines where parts of binary get loaded in memory
  - Only care about LOAD sections
- Load ‘memsz’ bytes at ‘offset’ into binary to the ‘vaddr’ (virtual address)

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00120 0x00120 R   0x4
  INTERP         0x000154 0x00000154 0x00000154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /usr/lib/libc.so.1]
  LOAD           0x000000 0x00000000 0x00000000 0x001a1 0x001a1 R   0x1000
  LOAD           0x001000 0x12345000 0x12345000 0x00b8a 0x00b8a R E 0x1000
  LOAD           0x002000 0x12346000 0x12346000 0x00440 0x00440 R   0x1000
  LOAD           0x002f90 0x12347f90 0x12347f90 0x0007c 0x0007c RW  0x1000
  DYNAMIC        0x002f90 0x12347f90 0x12347f90 0x00070 0x00070 RW  0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
  GNU_RELRO      0x002f90 0x12347f90 0x12347f90 0x00070 0x00070 R   0x1
```

Final steps

- Get Entry point from file header and return it
- Use entry point in ‘execp’

```c
uint32_t entry = hdr->e_entry;
    if (!elf_read_phdrs(addr, hdr->e_phoff,
        hdr->e_phentsize, hdr->e_phnum)) {
        cwrites( "ELF: Error reading program headers!\n" );
        return 0;
    }
    return entry;
```

```c
uint32_t elf_entry = elf_load_program(entry);
    if (!elf_entry) {
        __sprint( b256, "*** execp(): could not load binary at
address: %x\n",
                entry );
        PANIC( 0, b256 );
    }
    // Set up the new stack for the user.
    context_t *ct = _stk_setup( curr->stack, elf_entry, args );
```

Future Work

- This is a naive implementation
- Dynamic library support
    - Allow for relocations