Modular Kernel

Joe Marchione, Eli Sepulveda,
Sarah Dziobak, Lucas Sirbu,
Brendon Wolfe

CSI 4500/5500

Layout

The Problem

Creating a low resource modular kernel

- Streamlined basic kernel for simple OS functionality

Relevance

- Low-Level OS Concepts
- I/O configuration, System Interrupts,
- ISO Configuration from Assembly

Long-Term Goal Progress Report

- Interrupt Handling
- Adding backspace capability

Design Challenges

Pipeline Optimization

- Running and compiling the OS was inefficient as it took multiple files to configure the kernel
- Optimizations included GitHub reconfiguration for better organization, and file improvements for faster runtime

File Systems

- The base fat32 file system is included, however it took multiple iterations and research in order to successfully implement

Debugging Tools

- Implementing debugging tools proved to be a challenging addition
- Worked on designing tools to assist with bug related issues such as compilation, unfortunately timing issues limited successful implementation

Research Transition

Issues with scope

- Wide scope with limited time frame meant harder-to-reach longer-term goals
- Research revealed true extent of the scope, and our attention turned to researching topics in depth for future implementation

Resources

- Research revealed valuable supplemental information regarding OS Kernel construction
- Youtube as well as Github contained structured pre-made examples along with source code

Transition to Research

- Group transitioned to researching full scope as well as researching practical implementations

Tools and Resources

Youtube Tutorial

- https://www.youtube.com/watch?v=9t-SPC7Tczc

GitHub

- GitHub - nanobyte-dev/nanobyte_os at videos/part1

Website Tutorial

- https://medium.com/@mckev/create-our-own-kernel-2902a68b062b

Lessons Learned

Multiboot kernel

- Our initial iteration of this project used a multiboot loader.
- This is difficult for adding additional kernel features
- Not ideal for beginners

BIOS for bootloading

- Our final project uses BIOS for bootloading as this directly supports booting from filesystems and interrupts
- Adaptable and efficient for beginners

Optimizations

- Initially we should have started with BIOS for bootloading due to its adaptability
- Downloading vscode within our VM is the most efficient way to develop this project

Lessons Learned

Choosing Compilers

- Choosing a compiler for our OS is incredibly important.
- Different compilers offer varying levels of optimization, debugging support, as well as compatibility with our target architecture.
- Some may generate machine code efficiently, using less memory and executing quicker while others may offer more robust debugging tools.

Custom printf

- No access to libc requires us to implement our own basic functions.
- When creating a simple function like printf we need to implement everything from the ground up including buffers to handle number conversions, format strings for flexibility, newline handling for clarity, etc.

Custom File System

- The file system also needs to be built from the ground up.
- Building the researched FAT file system requires various components including: the boot sector, FAT table, root directory, file handles, and basic file mechanisms of opening, reading, and closing a file.

Lessons Learned

Memory Functions

- Having no way to access the memory is very limiting within all programs including an OS.
- This requires us to implement standard functions such as memcpy, memset, and memcmp to copy data, initialize memory regions, and compare blocks of memory.

String Functions

- More functions that need to be built for an OS include functions regarding strings.
- This allows for string handling operations like copying, searching and measuring length of strings. These are essential for working with user input, filenames, and error messages.

Makefiles

- When building an OS, there are a lot of files involved that all need to be compiled when modifications are made.
- Makefiles streamline this process by saving time, reducing the impact of human error, and ensures consistency when creating new builds of the OS.

Example Makefile

```makefile
TARGET_ASMFLAGS += -f elf
TARGET_CFLAGS += -ffreestanding -nostdlib
TARGET_LIBS += -lgcc
TARGET_LINKFLAGS += -T linker.ld -nostdlib

SOURCES_C=$(wildcard *.c)
SOURCES_ASM=$(wildcard *.asm)
OBJECTS_C=$(patsubst %.c, $(BUILD_DIR)/stage2/c/%.obj, $(SOURCES_C))
OBJECTS_ASM=$(patsubst %.asm, $(BUILD_DIR)/stage2/asm/%.obj, $(SOURCES_ASM))

.PHONY: all stage2 clean always

all: stage2

stage2: $(BUILD_DIR)/stage2.bin

$(BUILD_DIR)/stage2.bin: $(OBJECTS_ASM) $(OBJECTS_C)
	@$(TARGET_LD) $(TARGET_LINKFLAGS) -Wl,-Map=$(BUILD_DIR)/stage2.map -o $@ $^ $(TARGET_LIBS)
	@echo "--> Created  stage2.bin"

$(BUILD_DIR)/stage2/c/%.obj: %.c
	@mkdir -p $(@D)
	@$(TARGET_CC) $(TARGET_CFLAGS) -c -o $@ $<
	@echo "--> Compiled: " $<

$(BUILD_DIR)/stage2/asm/%.obj: %.asm
	@mkdir -p $(@D)
	@$(TARGET_ASM) $(TARGET_ASMFLAGS) -o $@ $<
	@echo "--> Compiled: " $<

clean:
	@rm -f $(BUILD_DIR)/stage2.bin
```

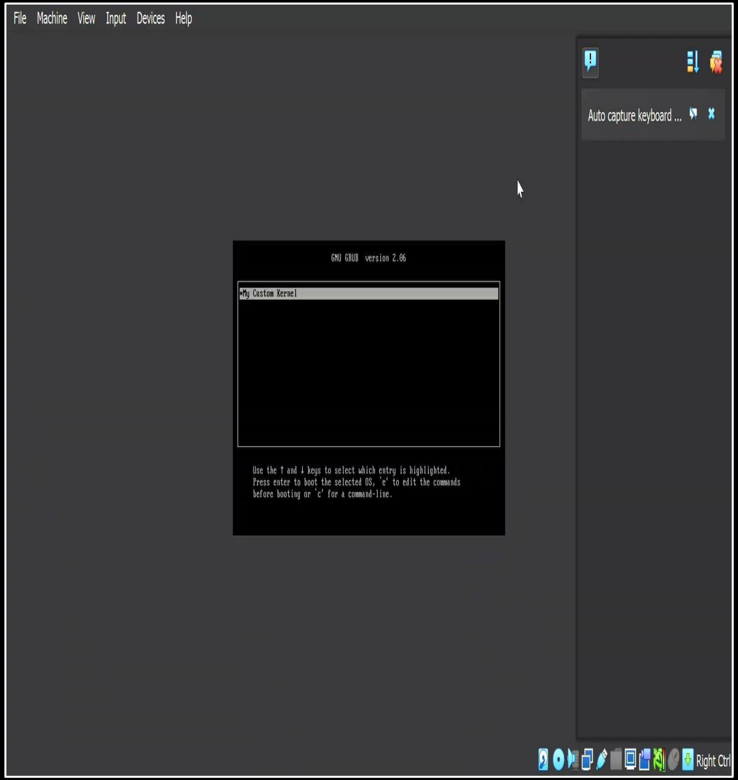
Memory Functions Implementation

```c
#include "memory.h"

void* memcpy(void* dst, const void* src, uint16_t num)
{
    uint8_t* u8Dst = (uint8_t *)dst;
    const uint8_t* u8Src = (const uint8_t *)src;

    for (uint16_t i = 0; i < num; i++)
        u8Dst[i] = u8Src[i];

    return dst;
}

void * memset(void * ptr, int value, uint16_t num)
{
    uint8_t* u8Ptr = (uint8_t *)ptr;

    for (uint16_t i = 0; i < num; i++)
        u8Ptr[i] = (uint8_t)value;

    return ptr;
}

int memcmp(const void* ptr1, const void* ptr2, uint16_t num)
{
    const uint8_t* u8Ptr1 = (const uint8_t *)ptr1;
    const uint8_t* u8Ptr2 = (const uint8_t *)ptr2;

    for (uint16_t i = 0; i < num; i++)
        if (u8Ptr1[i] != u8Ptr2[i])
            return 1;

    return 0;
}
```

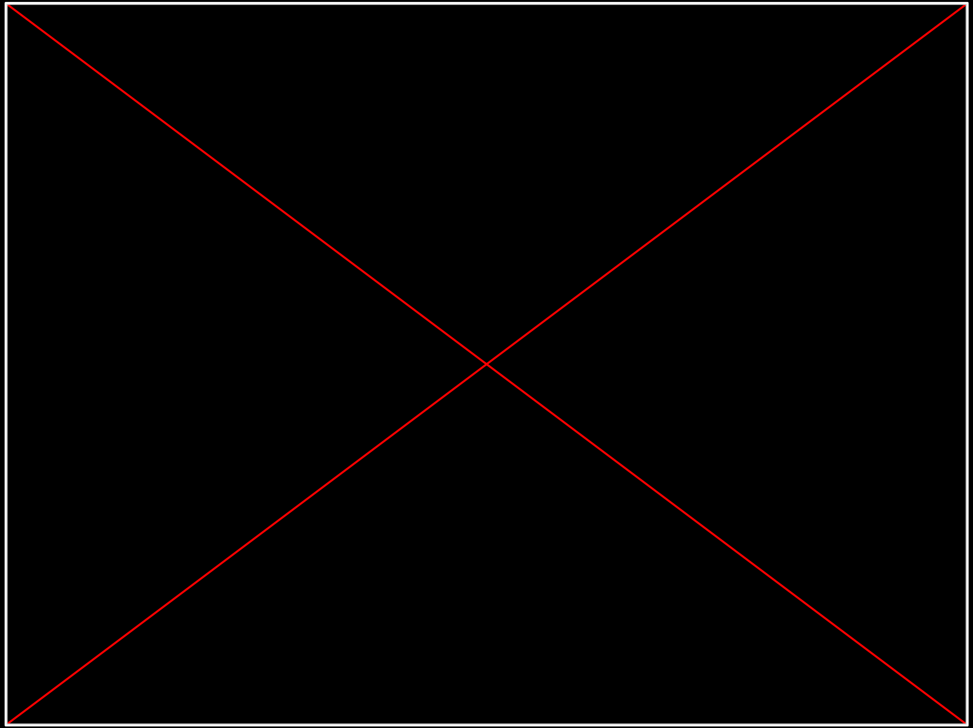
Part of the FAT File System Implementation

```c
bool FAT_Initialize(DISK* disk)
    // calculate data section
    uint32_t rootDirSectors = (rootDirSize + g_Data->BS.BootSector.BytesPerSector - 1) / g_Data->BS.BootSector.BytesPerSector;
    g_DataSectionLba = rootDirLba + rootDirSectors;

    // reset opened files
    for (int i = 0; i < MAX_FILE_HANDLES; i++)
        g_Data->OpenedFiles[i].Opened = false;

    return true;
}

uint32_t FAT_ClusterToLba(uint32_t cluster)
{
    return g_DataSectionLba + (cluster - 2) * g_Data->BS.BootSector.SectorsPerCluster;
}

FAT_File* FAT_OpenEntry(DISK* disk, FAT_DirectoryEntry* entry)
{
    // find empty handle
    int handle = -1;
    for (int i = 0; i < MAX_FILE_HANDLES && handle < 0; i++)
    {
        if (!g_Data->OpenedFiles[i].Opened)
            handle = i;
    }

    // out of handles
    if (handle < 0)
    {
        printf("FAT: out of file handles\r\n");
        return false;
    }

    // setup vars
    FAT_FileData* fd = &g_Data->OpenedFiles[handle];
    fd->Public.Handle = handle;
    fd->Public.IsDirectory = (entry->Attributes & FAT_ATTRIBUTE_DIRECTORY) != 0;
    fd->Public.Position = 0;
    fd->Public.Size = entry->Size;
    fd->FirstCluster = entry->FirstClusterLow + ((uint32_t)entry->FirstClusterHigh << 16);
    fd->CurrentCluster = fd->FirstCluster;
    fd->CurrentSectorInCluster = 0;

    if (!DISK_ReadSectors(disk, FAT_ClusterToLba(fd->CurrentCluster), 1, fd->Buffer))
    {
        printf("FAT: open entry failed - read error cluster=%u lba=%u\n", fd->CurrentCluster, FAT_ClusterToLba(fd->CurrentCluster));
        for (int i = 0; i < 11; i++)
            printf("%c", entry->Name[i]);
        printf("\n");
        return false;
    }

    fd->Opened = true;
    return &fd->Public;
}

uint32_t FAT_NextCluster(uint32_t currentCluster)
{
    uint32_t fatIndex = currentCluster * 3 / 2;
```

Keyboard handling



Custom printf



Potential Future Goals

File System

- Implementing a GUI based file system similar to stock linux
- Incorporating a file search system

Basic GUI

- Traditional Desktop for file storage/ program management
- Potential Windows like GUI to display programs and files

Module Loading

- Initializable Modules that add optional functionality on boot
- Incorporating modules into the file system for long term storage

Contributions

Eli Sepulveda	Module Loading, Boot Configuration
Joe Marchione	Keyboard handling, Bug Fixing
Sarah Dziobak	Bug Fixing, Keyboard Handling, Backspace capability
Lucas Sirbu	Bug Fixing, Presentation Coordination
Brendon Wolfe	Keyboard Handling, Boot Reconfiguration