Giovanni Coppola, Greg Leitkowski, Ezequiel Salas
CSCI-452

AHCI Description

The AHCI Module finds SATA Drives on the computer and identifies their model number to determine if they are the boot drive manufactured by Biwin or the HDD manufactured by Western Digital. If the DO_AHCI flag is defined in kdefs.h, the module is initialized in the kernel just after the kreport is printed.

The init method first installs an ISR for AHCI interrupts. Then, the PCI Bus is searched for an AHCI controller and the HBA memory space is found. This memory is then cast to be a pointer to an HBA_MEM struct.

The HBA_MEM struct is a structure to match the series of 32 bit registers of the HBA Memory space. Of particular interest are the first 4 register offsets, capabilities (0x0), global host control (0x4), interrupt status (0x8), and ports implemented (0xC).

This HBA_MEM struct is then passed to the probe_port() method where each port of the controller is searched for a SATA drive. In this method, AHCI mode and interrupts are enabled in the ghc register of HBA_MEM then each of the implemented ports shown in the pi register of HBA_MEM are searched for a SATA drive. Ports are represented by the HBA_PORT struct which contains each of the 32 bit registers of the port including status, error information, interrupt information, etc. SATA drives are identified with the signature register of the port.

If the port contains a SATA drive, the port is rebased and then its model number is identified. Rebasing the port consists first of stopping command execution with bits in the command register of the port. Then, space is allocated in memory for the command header, command table, and FIS structure that registers in the port will point to so they can be accessed by the device during command execution. After all of this memory is allocated, command execution is reenabled on the port.

The command header memory location is pointed to by the clb field of the HBA_PORT struct. It is represented by the HBA_CMD_HEADER struct and contains information about that command to be executed like the size of the FIS, its type, and whether to interrupt. The cbta field of this struct points to the base address of the command table. The command table is a struct representing the table of commands the device can execute. Commands are executed through FIS structures. The structure used by the ATA_CMD_IDENTIFY (0xEC) command is represented by the FIS_REG_H2D struct. In this struct is the fis_type, which is identified in the FIS_TYPE enum containing the different FIS types that can be used, whether the FIS is a command, and memory addresses to return data. However, the ATA_CMD_IDENTIFY returns data to the base address of the PRDT_ENTRY of the command table. The PRDT_ENTRY struct contains its base address in memory, its size, and whether the command should interrupt on completion.

After rebasing the port, the check_WD() method is executed, which ATA_CMD_IDENTIFY is sent to the port to retrieve a 512 byte data structure that contains the model number of the drive. The start of this model number is then printed at the end of the function.

The command is executed by first enabling all interrupts and clearing current interrupt statuses. Then, an available command slot is found that the command can be sent through. Then, the command information is populated into the command header and table. This includes allocating memory for a 512 byte buffer that the command will send data to. After command info is populated, the code waits for the drive to be free to accept a new command. Then, the command is issued by setting the bit corresponding to the command slot in the ci register of the port. The code then waits for the proper bits to be flipped to signify the command execution is completed. Then the response address is read from and the first two characters of the model number are printed.

Parallel Printer Description

The Parallel Printer module will establish a connection between the printer and the computer. It does so by performing I/O via a set of 3 ports from 0xD010, 0xD011, and 0xD012, the data register, status register, and control register respectively. First it initializes the printer with a PRIME signal to the control registers, this makes the printer clear out its buffer. It checks for 3 status codes from the status register at the start to return some type of output to the user. It will check for a BUSY signal which means that the printer is in one of 4 possible states but for the sake of this example we will mostly ignore the possible states. If it’s busy we loop until the status of the printer changes. After that we check for a PAPER OUT signal, if that is true we loop again letting the user put in paper and fix the problem. Last status we check is a generic ERROR code, if that is true then something else has occurred in the printer and it needs further examination. In that case we print out the status code we received in hex, inform the user of an error and return from the function. If all goes well, we then send out a collection of 4 bytes to the printer via the data register each one followed by a pulse of the STROBE signal sent to the control register and a delay to try to match the printer clock. This tells the printer that there is something to read from the data registers and gives it time to read it in, process it, and then return an ACKNOWLEDGE signal meaning the printer is ready to read in more data. As is, this timing is just off so on occasion the printer may receive one or more bytes of characters. That is the current implementation but the further goal was to take in keyboard input and print that out however since printing more than one character at a time is a challenge it would take a bit more time.

File System Description:

The terminal gets brought up with the menu to use the operating system. The user will enter an ‘i’ to initialize the file system. What this does is it will call the functions to allocate the linked lists in memory. The linked lists are a struct defined as such:

```
	typedef struct list_s {
		void *data;
		struct list_s *next;
	} list_s;
```

The data pointer will point to either an inode or a data block, depending on what is being initialized. There are 4 of these lists in the file system itself. To show this, this is the definition of the file system:

```
	typedef struct FileSystem_s {
		uint32_t num_free_blocks;
		uint32_t num_free_nodes;
		list_s *free_nodes;
		list_s *used_nodes;
		list_s *free_blocks;
		list_s *used_blocks;
		inode_s *current_inode;
		inode_s *previous_inode;
	} FileSystem_s;
```

There are 4 different linked lists in memory. Those are for the free data blocks, the used data blocks, the free inodes, and the used inodes. There are also pointers to the current inode in use, and the previous inode that was being used. The previous is set to NULL if we are still in the first working directory. The number of free blocks and free nodes also gets decremented once a pointer has been popped off the respective free list.

An inode has the following structure:

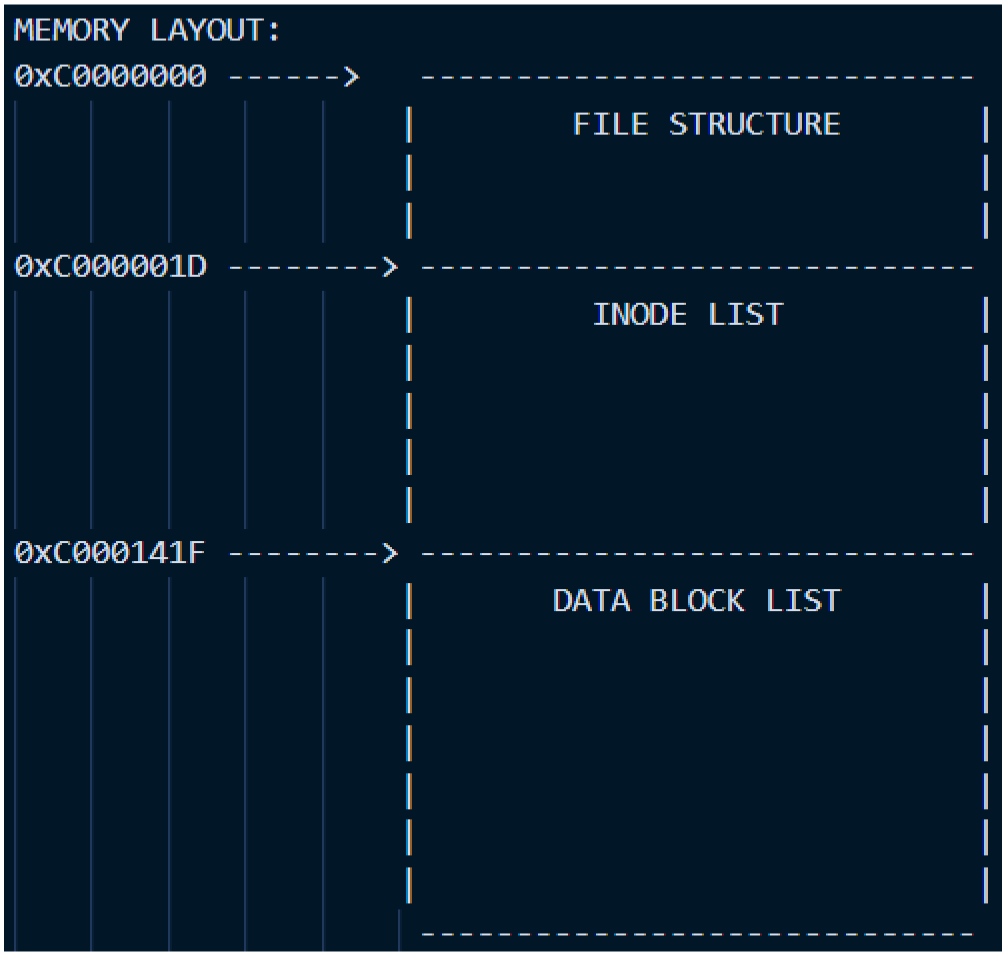
```
	typedef struct Inode_s {
		uint8_t size;
		uint8_t num_of_pointers;
		bool_t is_direct;
		void *direct[15];
		char name[MAX_NAME_LENGTH];
	} Inode_s;
```

The inode contains a size, that is not currently being used, a number of active pointers, a bool to represent if the first pointer is going to be directly to a data block (true) or another inode (false), an array of 15 pointers that will pointer to other inodes or data blocks respectively, and finally a name. What I would have changed about this in hindsight, would have been to separate the directories into their own data structure and then have an index to represent the pointer into the inode list that would contain the metadata about the directory/data block itself depending on what the current pointer would have been. And then that inode list would also have been in the file system. As for the data blocks, the structure is as follows:

```
typedef struct File_s {
        char name[MAX_NAME_LENGTH];
        char data_block[SIZE_OF_DATA_BLOCK];
} File_s;
```

The data block simply contains a string for the name, and a string of data that will be used like a file. I would have changed this to match the changes I would have made to the directories. So the file would also contain an index into the inode which would contain metadata about it.

For the memory layout, I have a simple image that will describe the memory layout of where the file system gets initialized:



The file system gets initialized at the start of the 4th gig in memory (which is the entire upper gig in our system). Once the file system is initialized, the linked list for the inodes will begin at 0xC000001D. This will contain all 64 inodes that can be used in the system. After that, will be the linked list of data blocks. There are 960 that can be used in total, so all 960 will be placed there. The nice thing about my implementation, is that the inodes and data blocks will stay in those sections regardless of which list it is part of. The addresses might be allocated a little out of order depending on the delete operations and create operations, but they will always fall within those blocks dedicated to the respective structure.