Review: On Disk Structures

At the most basic level, a HDD is a collection of individually addressable **sectors or blocks** that are physically distributed across the surface of the platters.

Older geometric based addressing is called CHS for cylinder-head-sector. This triple value uniquely identifies every sector.

CHS was straightforward for older disks which had the same number of sectors for every track.

Modern HDDs use zone recording to not waste space, so outer tracks have more sectors than inner tracks, which makes CHS addressing very complex.

New approach is to provide a disk controller interface that recognizes a simpler addressing scheme called LBA – logical block addressing – that simply numbers all blocks in sequence from 0, 1, 2, … up to the maximum. The controller translates LBA to CHS and uses CHS only internal to the disk itself.

Partitions

A HDD can be used as one enormous pool of sectors or blocks. With a small amount of additional organization, the disk can be used for multiple purposes.

The Master Boot Record and the Partition Table

If you are planning to use a HDD for a single partition file storage and no bootable OS, then technically you don't need a MBR. But if you plan to partition the disk, or if you want to make the disk bootable, then you'll generally need one.

The MBR is a small table near the physical beginning of the disk that contains

- The partition table
- First stage boot code

The partition table is a table of contents to each partition on the HDD. The first stage boot code is code that executes early in the boot process to assist the loading of the OS.

What's In a Partition?

Just like the HDD has its own table of contents in the MBR, each partition can have its own toc called a VBR – volume boot record. A volume is similar to a partition. The difference is that a partition usually means a raw division of a disk with no structure, while a volume is a partition that has been organized into a file system. The VBR may contain additional boot code that executes after the MBR boot code.

Building a File System: Turning a Partition into a Volume

A collection of raw sectors by itself is not a file system. A file system requires a set of data structures to manage tasks such as:

- Using pointers to locate directories starting with the root directory and descending downward through subdirectories as needed.
- Creating new files by finding free sectors, linking them together to provide space for the file, then storing a pointer to the first sector in its containing directory.

- Occasionally cleaning up the sectors by defragmenting them so that sectors making up a given file are not widely scattered across the disk.
- Occasionally running a consistency check to detect and hopefully repair problems like lost or damaged files.

For most file systems, the sector or block is too small to work with, so a group of sectors called a cluster is used instead.

Different file systems use very different data structures to organize clusters into files. Some of the most well-known:

- FAT: the File Allocation Table FS from Microsoft
- NTFS: the New Technology FS from Microsoft
- ReFS: the Resilient FS from Microsoft
- Ext3/Ext4: The External FS widely used by Linux
- YAFFS: Yet Another Flash File System, one of several FS designed for use with flash memory.

FAT File System Details

In the FAT, the volume is divided up as follows:

- VBR
- FAT
- FAT copy
- The root directory
- Free clusters

There are two kinds of file in the FAT FS:

- Regular file consisting of content distributed across clusters
- Directory file consisting of pointers to other files

The two data structures that keep everything organized are:

- The FAT table entry
- The directory entry

Most info about a file is stored in its directory entry. This holds:

- Name
- Directory or regular file flag
- Attributes: hidden, archive
- Timestamps: created, modified, etc.
- Size
- Pointer to first cluster for the file

Significantly, note that only the pointer to the first cluster is stored in the directory. If the file fits within one cluster, then this is all that's needed. But if the file extends across multiple clusters, more info is required. This is where the FAT table itself is used.

The FAT is a simple data structure that is indexed by cluster and contains a single status item for that cluster:

- Free/Unused
- If in use, indicates the next cluster in the chain, or end of chain marker.
- Bad/Unusable

The number of entries in the FAT must equal the number of clusters in the volume. For older smaller disks this wasn't an issue. But for larger disks, this becomes a problem, the FAT has to be enormous. This can be partially fixed by making the clusters larger and reducing their number. But this causes its own problems. Small files and large clusters result in wasted space called **slack** at the end of the last cluster, since file size must be rounded up to the next full cluster.

Instructions and Address Modes

Each instruction is broken down into several fields.

Opcode	**Operand 1**	**Operand 2**	**…**

Some complex instruction set CPUs (CISCs) might have 3-operand formats, for example.

During the fetch pipeline step, an instruction is fetched (moved) from the RAM address indicated by the program counter (PC) to the CPU.

During the decode step, the opcode of the fetched instruction is examined to dispatch the instruction to the proper hardware to execute it.

There may need to be multiple operand decode steps in the pipeline to get the actual operand value.

Within the operand fields, there are subfields for address mode and address

Address Mode	**Address**

The address cannot be properly interpreted unless we know the address mode. Here are some common examples of address mode and their meaning:

Immediate	The address contains the **actual** operand
Direct	The address is the RAM address of where the actual operand is stored
Indirect	The address is the RAM address of a pointer to where the actual address is stored
Register	The address identifies a CPU general purpose register that contains the operand
Base + Index	The address is indicated by some combination of base value in a register plus index or offset in another register

There is quite a bit of variation on address mode options for different CPUs.

How an HLL Source File Becomes an Application

When an application is launched, it becomes a process that is managed by the OS.

A **process image** is created that contains all code and all data required by the running app.

Initially the image is stored on disk in the swap space.

To begin execution, some initial allocation of pages must be brought into RAM by the virtual memory system.

The PC (program counter) register in the CPU is then loaded with the address of the starting instruction and the process begins to execute.

The process image contains several regions holding different kinds of information, all parts will need to be paged into RAM at some point:

- Global Variables
- Code
- Stack and Heap

The **heap** is used for dynamically allocated memory. In C, malloc() allocates memory from the heap, it remains allocated until it is freed later with the free() function. In Java, malloc() is replaced by “new”, and free() is replaced by the automatic garbage collector.

The **stack** is used to create space for local variables for each function call. Local variables are created on the stack when the function is called, and are deleted when the function returns. This process works across multiple function calls, even recursive calls. Each function call has its own region of the stack that is separate from the stack used by any other function call.

The heap can become fragmented because allocation and free operations are not required to be in any sequence. The stack can never be fragmented because it is allocated and freed in strict nested order.

Process Image

- When an application is launched, a process image must be created in swap space
- Once the process image is laid out, the OS can begin to move it into RAM one page at a time
- The CPU issues addresses for the process image that refer to the process's unique logical address space
- The VM (virtual memory) system will translate logical page numbers to physical page frame numbers.
- Each process has its own process image with its own logical address space
- Each process image is divided into segments to hold the following info for that process:
    - Global variables
    - Code (instructions)
    - Data (stack + heap)
- The stack is used to manage function calls
- Each function call gets a chunk of stack called a stack frame or call frame
    - The call frame holds local variables plus passed parameters and return values from the calling function.
    - Frames are created automatically when a function is called
    - Frames are deleted automatically when a function returns
- Because of this behavior, the stack does not experience fragmentation. Memory is returned to the stack in LIFO (last in first out) order, so there's no opportunity for holes or fragments of unusable stack space to develop.

The Heap

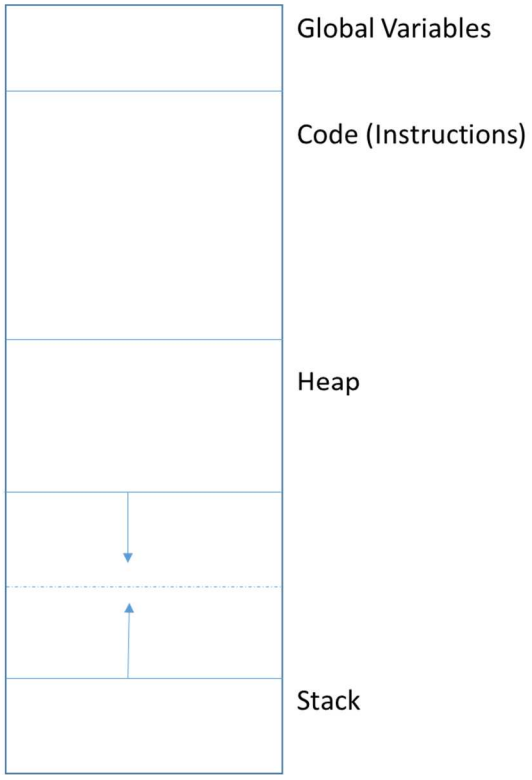
The heap is used to manage dynamically allocated memory.

- In C:
    - The **malloc**() function takes a chunk of memory from the heap and allocates it to the process.
    - The **free**() function gives a chunk of memory back to the heap so that it can be reused.
- In Java:
    - The **new** operation allocates space from the heap
    - The **garbage collector** automatically returns space to the heap, no need to call **free**()
- Heap management is more complex than stack management.
- On the stack, memory allocation and deallocation is strictly nested by function calls. The stack cannot become fragmented.
    - Deallocation of call frames occurs in exactly reverse order of allocation.
    - All allocated space is reclaimed without the use of complex management algorithms.
- Unlike the stack, the heap can become fragmented, because deallocation does not happen in precisely reverse order of allocation.
    - Chunks are allocated and deallocated in a more unpredictable order, leaving the heap a mix of used and unused space.
    - The garbage collector can also defrag the heap occasionally to make sure the heap can effectively reuse its space
    - Programs with poor heap management (incorrect use of malloc()/free()) can crash when the heap runs out of space, this is called a memory leak.
    - Memory “leaks away” as it is allocated and never returned properly to the heap when it is no longer needed

Process Image

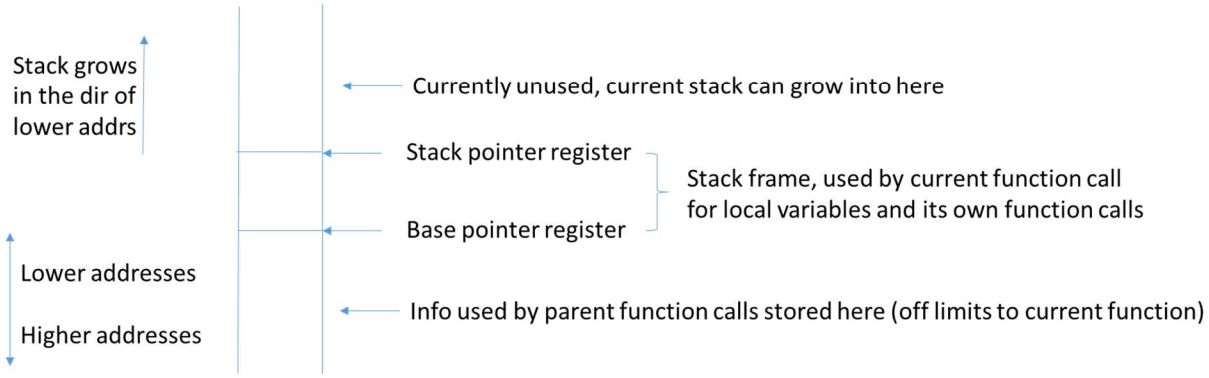
Any address generated by CPU for this process must refer to a location in this image.

There is no global memory that is shared between process images
Two active processes that wish to exchange information must do so through I/O mechanisms such as message passing or pipes.



The Stack

- Dynamic data structure to hold all data local to a function call
  - Local variables
  - Passed parameters to and return values from new function calls
- Each process image has its own stack in its own logical address space



More Detail About the Stack

To explain how the stack is used when a program is running, let's look at a simple example application based on the following C source code:

```
#include <stdio.h>
#include <stdlib.h>

int a, b, c;                    // global variables

int f() {                       // function f()
      int x;                    //    local variables, local to f
      int y;
      int z;

      x = 3;
      y = 7;
      z = x + y;

      return z;
}

int main(void) {                // function main
      int p;                    //    local variables, local to main

      p = f();                  // function call:  main calls f

      return 0;
}
```

This program consists of 3 global variables – a, b, and c – and two functions – main() and f(). main() is the entry point function with a single local variable p, and f has three local variables – x, y, and z. main() calls f(), stores the return value in p, and returns.

Below is the assembly language. Note the following:

- Global variables are allocated in a separate segment of the process image before the start of the text segment (code or instructions).
- The assembler places names of all variables into a symbol table to be used while code is being generated. But variable names are all converted to address modes in the final assembly language output.
- Global variables are stored at absolute addresses.
- Local variables including passed function params and return values are stored on the stack at relative offsets to the stack pointer.

The program converted to x86 assembly language:

```
; Microsoft (R) Optimizing Compiler Version 19.00.23026.0

_DATA SEGMENT
      COMM  _a:DWORD                              ; a, b, c:  global variables
      COMM  _b:DWORD
      COMM  _c:DWORD
_DATA ENDS

PUBLIC      _f
PUBLIC      _main

_TEXT SEGMENT
      _p$ = -4                                    ; p:  local variable for main

      _main PROC
            push  ebp
            mov   ebp, esp
            push  ecx
            call  _f
            mov   DWORD PTR _p$[ebp], eax
            xor   eax, eax
            mov   esp, ebp
            pop   ebp
            ret   0
      _main ENDP
_TEXT ENDS

_TEXT SEGMENT
      _z$ = -12                                   ; x, y, z:  local vars for f
      _y$ = -8
      _x$ = -4

      _f    PROC
            push  ebp
            mov   ebp, esp
            sub   esp, 12
            mov   DWORD PTR _x$[ebp], 3
            mov   DWORD PTR _y$[ebp], 7
            mov   eax, DWORD PTR _x$[ebp]
            add   eax, DWORD PTR _y$[ebp]
            mov   DWORD PTR _z$[ebp], eax
            mov   eax, DWORD PTR _z$[ebp]
            mov   esp, ebp
            pop   ebp
            ret   0
      _f    ENDP
_TEXT ENDS
END
```

Stack Operation

The stack is a key structure in memory for the execution of any HLL program that has been translated to machine language. The typical CPU contains at least one and possibly additional special purpose registers dedicated to maintaining the state of the stack. In the case of the x86 compatible machine language used here, there are two registers to manage the stack:

- ebp: base pointer
- esp: stack pointer

The base points to the bottom of the stack from this function's perspective. There is information on the stack beyond this point, but it should not be touched by this function because it represents state information stored there by other still-active functions. Erroneously modifying data below the stack base would be the same as randomly changing the values of variables being maintained by other functions in the program.

The stack pointer points to the top of the stack. It can grow as needed based on the use of local variables and function calls by the current function. The base of the stack is in the higher address end of memory, and it grows toward lower address memory.

Function Calls and the Stack

Whenever one function calls another function, the stack is required to make the communication between the functions work correctly. The stack performs two important roles for each function call in an application:

- Storage for local variables for the current function
- Storage for passed parameters and return values between calling and called function

A block of storage called a stack frame or call frame is created on the stack every time a function is called. This is true even in the case of a recursive function for which a single function is called multiple times and all function calls are active at the same time.

Global variables have a special area of memory reserved for them and this storage exists for the duration of the application execution.

Local variables in contrast are only created when the function is called. At that time they are created on the stack. They are deleted or erased when the function returns.

Function Execution Preamble

Code at the beginning of every function makes room on the stack for its local variables. Here is the preamble for function f():

```
push   ebp
mov    ebp, esp
sub    esp, 12                          ; 0000000cH
```

The first thing the preamble does is to save the old value of the stack base by pushing it onto the stack. It then moves the stack pointer up to reserve space for this function's local variables. In this case there are 3 integers, each occupies 4 bytes, so the stack pointer must be moved down by 12 bytes.

In the assembly language program, there is no explicit symbol to represent the names of local variables, they are accessed by their location relative to the stack pointer. So when the HLL program uses a local variable, the compiler translates the reference to the named local variable into an operand that uses stack relative addressing. Each local variable is a location on the stack at a unique offset from the stack base.

Function Execution Cleanup

Code at the end of every function cleans up the stack to put it back into the state it was in before the function was called:

```
mov    esp, ebp
pop    ebp
```

Once the function ends, the space for the local variables are deleted from the stack, and then the old value of the stack base is popped from the stack to restore it to the ebp register.

Referencing Local Variables in Assembly

Source Code

```
x = 3;
y = 7;
```

Assembly: The first three lines are simply aliases that introduce symbols for each local variable based on their location on the stack. The last two lines are the actual code to implement the assignments.

```
_z$ = -12                                   ; size = 4
_y$ = -8                                    ; size = 4
_x$ = -4                                    ; size = 4

      mov   DWORD PTR _x$[ebp], 3
      mov   DWORD PTR _y$[ebp], 7
```

In this example, suppose we want to store 3 into local variable x. The variable x is located at an offset of -3 from the stack base. The symbol "_x$" is aliased to the value "-4." So the instruction to store 3 into x is:

```
      mov   DWORD PTR _x$[ebp], 3
```

Or if we substitute the symbol for its value:

```
      mov   DWORD PTR -4[ebp], 3
```

This addressing mode expresses the operand address as an offset from the content of the ebp (stack base) register. In English, this statement says:

> Take the value 3 and store it into the location at offset -4 from the stack base.

This is a typical approach for most compilers and CPUs to allow local variables to be properly addressed in assembly language.

Function Calls

The final usage of the stack by an application is when one function calls another function. The stack is the missing link that allows parameters to be passed from the calling function to the called function, and for the called function to return a value back to the calling function.