**Case study of Memory Management on Linux, Windows & MacOS**

**INTRODUCTION**

* The Memory Management System is one of the important core parts of an operating system.
* It is the functionality of an operating system which manages primary memory and moves processes back and forth between main memory and disk during execution
* The essential requirement of memory management is to provide ways to dynamically allocate portions of memory to programs at their request and free it for reuse when no longer needed.
* We will be comparing the Memory Management (MM) Sub-Systems of these operating systems - Linux, Windows and MacOS since these are very popular operating systems for use as a desktop especially with beginners, and has now evolved into a mature operating system.

**VIRTUAL MEMORY**

**LINUX**

* Maintains two separate views of a process’s address space. Logical view and physical view.

**Logical view**

* Logical view consists of a set of non overlapping regions, each region representing a continuous, page-aligned subset of the address space.
* Each region is described internally by a single vm\_area\_struct structure.
* Vm\_area\_struct structure defines the properties of the region, including process’s read, write, and execute permissions.
* Some of the fields in this structure are:
  + - * vm\_start
      * vm\_end
      * vm\_file
* Uses balanced binary tree for the fast lookup of the region.

**Physical view**

* This view is stored in the hardware page tables for the process.
* It is managed by a set of routines which are invoked from kernal’s software-interrupt handlers.
* Each vm\_area \_struct in the address space contains a field pointing to a table of functions that implement the key page-management functionality for any given virtual memory region.

**Virtual memory regions**

* One property that characterizes virtual memory is the backing store for the region
* A region backed by nothing is called ‘demand-zero memory’.
* When a process tries to read a page from such region it is simply given back a page of memory filled with zeroes.
* Virtual memory regions are also defined by its reactions to writes.
* If a process writes to privately mapped region then the pager detects copy-on-write is necessary.
* Writes to shared region results in updating of the object mapped into that region so that change will be visible immediately to any other process that is mapping that object.

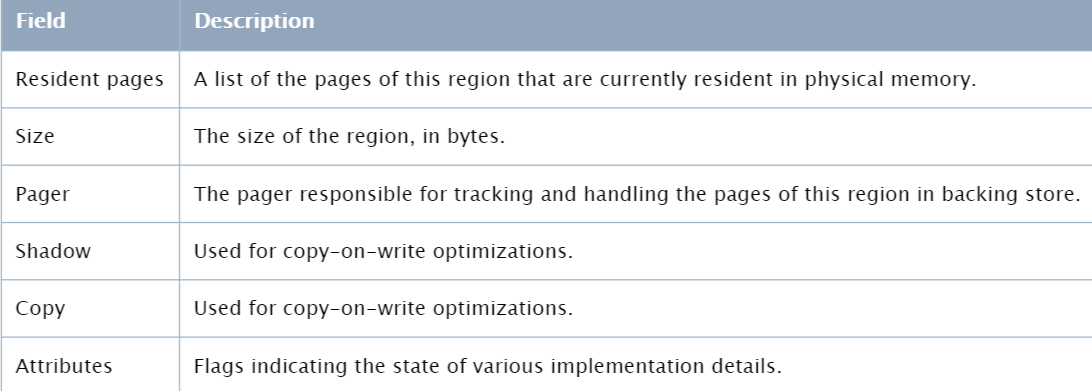
**WINDOWS**

* The virtual memory (VM) manager in windows uses a page-based management. scheme with page sizes of 4KB and 2MB on AMD64 and IA-32-compatible processors and 8KB on the IA64.
* Pages of data allocated to a process that are not in the physical memory are either stored in the ‘paging files’ on the disk.
* A page can be also marked as zero-fill-on-demand, which initializes the page with zeros before it is allocated, thus erasing the previous contents.
* On IA-32 processors, each process has a 4-GB virtual address space out of which upper 2GB are used by the windows in kernel mode to access the operating-system code and data structures.
* For the AMD64 architecture windows provides a 8-TB virtual address space.
* The windows VM manager uses a two-step process to allocate virtual memory.
* The first step reserves one or more pages of virtual address in the process’s virtual address space.
* The second step commits the allocation by assigning virtual memory space.
* Windows limits the amount of virtual memory space a process consumes by enforcing a quota on committed memory.
* A process de-commits memory that is no longer required.
* Windows implements shared memory by defining a ‘section object’.
* After getting a handle to section object a process maps the memory of the section to a range addresses, called a view.
* Windows allows sections to be mapped not just into the current process but into any process for which the caller has a handle there by allowing sharing.

**MacOS**

* Both OS X and iOS include a fully-integrated virtual memory system.
* Both the systems provide up to 4GB of addressable space per 32-bit process.
* In addition, OS X provides approximately 18 exabytes of addressable space for 64-bit processes.
* In case OS X it supports backing store (portion of the disk that stores the unused data) but iOS doesn’t.
* Unlike most UNIX-bases operating systems OS X doesn’t use a pre-allocated disk partition for the backing store. Instead it uses all of the available space on the machine’s boot partition
* In OS X kernel associates a VM object with each region of the logical address space.
* The kernel uses VM object to keep track and manage the resident and non-resident pages of the associated regions.

The following are the some of the fields in the VM object.



**HANDLING PAGE FAULTS**

**LINUX**

* Page faults are triggered by CPU and handled in the page\_fault\_handler.
* When the page fault handler is invoked, it first needs to determine whether the page fault is caused by an access to a valid page. If not, the page fault handler simply sends a segmentation violation signal to the faulting process and returns.
* Otherwise it takes one of several possible actions:
* If it is a demand page fault (page is accessed for the first time) then handler allocates a new page frame and initializes it.
* If the page has been paged out to swap space, the handler reads it back from disk into a newly allocated page frame.
* If the page fault occurred due to a page fault based optimization then the handler takes the appropriate recovery action.

**WINDOWS**

* When a page fault occurs the kernel trap handler dispatches this kind of fault to the memory manager fault handler to resolve.
* This routine runs in the context of thread that incurred the fault and is responsible for attempting to resolve the fault or raise an appropriate exception.
* These faults are caused due to variety of conditions as follows

|  |  |
| --- | --- |
| **REASON FOR FAULT** | **ACTION TAKEN** |
| Accessing a page that isn't resident in memory but is on disk in a page file or a mapped file | Allocate a physical page, and read the desired page from disk and into the working set |
| Accessing a page that isn't committed | Access violation |
| Writing to a page that is read-only | Access violation |
| Accessing a page from user mode that can be accessed only in kernel mode | Access violation |

**MacOS**

* There are two kinds of page faults.
* Soft fault
* Hard fault
* A soft fault occurs when the page of the referenced address is resident in physical memory but is currently not mapped into the address space of this process.
* A **hard fault** occurs when the page of the referenced address is not in physical memory
* For soft faults, the kernel maps the physical memory containing the pages to the virtual address space of the process. The kernel then marks the specific page as active
* For hard faults, the VM object’s pager finds the page in the backing store or from the file-mapped file, depending on the type of pager. And then the pager moves the page into physical memory and places the page on the active list.

**-----------------------------------------------------------------------------------------------------------------------**

The latest operating systems such as Windows, Linux, FreeBSD, and Mac OS X use the demand paging whereas classic Mac OS until Mac OS 9 use the **segmentation**.

**DEMAND PAGING**

* One way to save physical memory is to only load virtual pages that are currently being used by the executing program.
* Pages that are never accessed are thus never loaded into physical memory. A demand-paging system is similar to a paging system with swapping.
* The device to move data between hard disc and memory is a **pager**. The pager responds to page faults and manages their replacement.

**LINUX**

* An executable image has been memory mapped into a processes virtual memory it can start to execute.
* If page referenced is not there in physical memort the processor will report a page fault to Linux. The page fault describes the virtual address where the page fault occurred and the type of memory access that caused.
* Linux differentiate between pages that are in the swap file and those that are part of an executable image on a disk somewhere.

**WINDOWS**

* The Virtual Memory manager of the operating system use special Paging techniques namely **Disc Paging** and **Demand Paging** to overcome the space limitation of Physical memory.
* Windows NT is able to drop the segmentation architecture of previous versions of Windows.
* **Disc Paging** extends the computer’s physical memory (RAM) by reserving space on the hard disc called **Page File** which the processor views as non-volatile RAM.
* **Page file** is a reserved portion of a hard disk that is used as an extension of RAM for data in RAM that hasn't been used recently.
* The MMU generates a **Page fault** if the virtual page address has a missing frame number in the page table OR the frame number does not exist in physical memory OR the physical address is not part of the current **working set**.

**MAC**

* A VM object is an abstraction for the contiguous data that can be mapped into a region of an address space.
* Mac OS X includes two built-in pagers: the **default pager** and the **vnode pager**. They are used by VM system to actually get data into the VM objects.
* The VM object maps regions in the backing store through the default pager and maps file-mapped files through the vnode pager.

**PAGE FAULT**

* A memory access fault occurs when code tries to access data at a virtual address that is not mapped to physical memory. There are two kinds of faults:
* A soft fault occurs when page referenced address is resident in physical memory but currently not mapped into address space of process
* A hard fault occurs when page referenced is not in physical memory, but may swapped out into backing store.

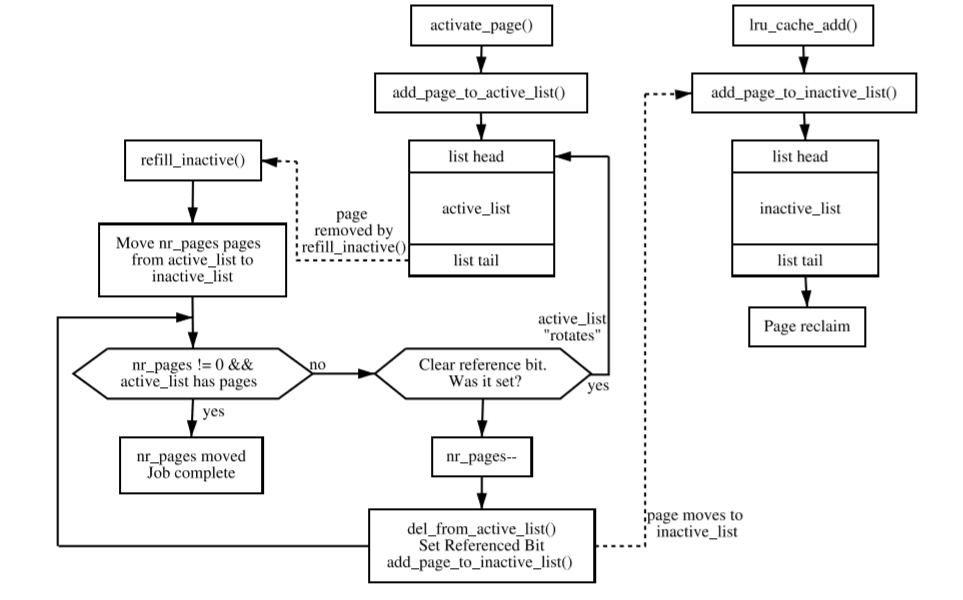
**PAGE REPLACEMENT**

* Page replacement algorithms are the techniques using which an OS decides which memory pages to swap out, write to disk when a page of memory needs to be allocated.
* Whenever a page fault occurs if free page is not available then pages in physical memory need to be swapped out.

**MAC OS**

* Mac OS X adopts a **second-chance first in, first out (FIFO) algorithm.**
* If a page is not been referenced for long time instead of immediately expelling that page from the main memory, the algorithm send the pages to the inactive queue. . The kernel maintains and queries three system-wide FIFO lists of physical memory pages:
* The active list contains pages that are currently mapped into memory and have been recently accessed.
* The inactive list contains pages that are currently resident in physical memory but have not been accessed recently.
* The free list contains pages of physical memory that are available for allocation.
* Page replacement is performed by a special kernel thread called the **pageout daemon**.
* The kernel continuously compares the number of physical pages in the free list against a threshold to balance the queues. Page replacement occurs by removing pages from the inactive list, placing them on the free list when the number of pages in the free list dips below this threshold.
* The inactive list serves as a second chance for pages about to be replaced.

**LINUX**



* Linux keep active list about two-thirds the size of the total page cache.
* Linux takes a simpler approach by using **refill\_inactive()** to move pages.
* When pages reach the bottom of the list, the referenced ﬂag is checked. If it is set, it is moved back to the top of the list, and the next page is checked. If it is cleared, it is moved to top of inactive list.

**WINDOWS**

* Windows uses a per-working-set, least-recently-used (LRU) replacement policy to take pages from processes. It approximates CLOCK algorithm.
* When a process is started, it is assigned a default minimum working-set size.
* The working set of each process is allowed to grow until the amount of remaining physical memory starts to run low, at which point the VM manager starts to track the ***age*** of the pages in each working set.
* **Age** of a page is how long it has been last referenced. VM manager trims the working set to remove older pages.
* Age is determined by periodically making a pass through the working set of each process and incrementing the age for pages that have not been marked in the PTE as referenced since the last pass.
* A process can have its working set trimmed even when plenty of memory is available, if it was given a **hard limit** on how much physical memory it could use.

**COPY-ON-WRITE**

* Copy-on-write page protection is an optimization the memory manager uses to conserve physical memory.
* A process maps a copy-on-write view page , instead of making a process private copy at the time the view is mapped, the memory manager defers making a copy of the pages until the page is written to.
* If a thread in either process writes to a page, a memory management fault is generated.

**WINDOWS**

* The memory manager sees that the write is to a copy-on-write page, so instead of reporting the fault as an access violation, it allocates a new read/write page in physical memory, copies the contents of the original page are written to new page.
* Copy-on-write is one example of an evaluation, lazy evaluation that the memory manager uses as often as possible. Lazy-evaluation algorithms avoid performing an expensive operation until absolutely required.

**LINUX**

* Linux recognizes a COW page because, even though the PTE is write protected, the controlling VMA shows the region is writable. It uses the function **do\_wp\_page()** to handle it by making a copy of the page and assigning it to the writing process.

**MAC**

* A VM object may map regions to another VM object.
* If the VM object is involved in a copy-on-write (vm\_copy) operation, the shadow and copy fields may point to other VM objects. Otherwise both fields are usually NULL.

**MEMORY ALLOCATION**

* Memory allocation is a process by which computer programs and services are assigned with physical or virtual memory space.
* It is the process of reserving a partial or complete portion of computer memory for the execution of programs and processes. Memory allocation is achieved through a process known as memory management.
* Memory allocation is primarily a computer hardware operation but is managed through operating system and software applications. Memory allocation process is quite similar in physical and virtual memory management. Programs and services are assigned with a specific memory as per their requirements when they are executed. Once the program has finished its operation or is idle, the memory is released and allocated to another program or merged within the primary memory.
* Memory allocation has two core types:
* Static Memory Allocation: The program is allocated memory at compile time.
* Dynamic Memory Allocation: The programs are allocated with memory at run time.

**LINUX**

* Linux uses various APIs for memory allocation, few of them are listed below:
* kmalloc()
* vmalloc()
* kvmalloc() etc.
* **kmalloc():**
* kmalloc is the normal method of allocating memory for objects smaller than page size in the kernel.
* It is a kernel memory allocation function, such as malloc() in user space.
* It is allocated from the LOW\_MEM region.
* Memory returned by kmalloc is contiguous in physical memory and in virtual memory.
* kmalloc is declared in <linux/slab.h> header.
* The following is the prototype of kmalloc:



**Parameters:**

size\_t size: how many bytes of memory are required

gfp\_t flags: the type of memory to allocate

* + kmalloc family has various derivatives for memory allocation, few of them are:
* **kmalloc\_array():** to allocate memory for an array.
* **kcalloc():** the memory is allocated for an array and is set to zero initially.
* **kzalloc():** to allocate memory, initially set to zero.
* **kzalloc\_node():** to allocate zeroed memory for a particular memory node.
* **vmalloc():**
  + vmalloc is used for allocating memory for objects larger than page size.
  + The memory returned is contiguous only in virtual space (not physically contiguous).
  + The returned memory comes from HIGH\_MEM zone.
  + The memory returned by vmalloc cannot be used outside the microprocessor since we cannot assert upon memory being physically contiguous.
  + vmalloc is slower than kmalloc because it has to retrieve the memory, build page tables and sometimes has to remap into virtually contiguous range.
  + vmalloc is declared in <linux/vmalloc.h> header.
  + The following is the protoype for vmalloc:

**void \* vmalloc(unsigned long*size*)**

**Parameter**

unsigned long size: allocation size

* + Few derivatives from vmalloc family are:
    - **vzalloc():** allocate virtually contiguous memory with zero fill.
    - **vmalloc\_user():** to allocate zeroed virtually contiguous memory for userspace.
    - **vmalloc\_node():** to allocate memory on a specifc node.
    - **vzalloc\_node():** to allocate memory on a specific node with zero fill.
* **kvmalloc:**
  + If you are not sure whether the allocation size is too large for kmalloc, it is possible to use kvmalloc() and its derivatives. It will try to allocate memory with kmalloc and if the allocation fails it will be retried with vmalloc.
  + kvmalloc in declared in <linux/mm.h>.
  + Prototype of kvmalloc is:

**void \* kvmalloc\_node(size\_t*size*, gfp\_t*flags*, int*node*)**

attempt to allocate physically contiguous memory, but upon failure, fall back to non-contiguous (vmalloc) allocation.

**Parameters:**

size\_t size: size of the request.

gfp\_t flags:gfp mask for the allocation - must be compatible (superset) with GFP\_KERNEL.

int node: numa node to allocate from.

* kvfree() is used to free the memory allocated by kmalloc(), vmalloc() or kvmalloc().

**WINDOWS**

* APIs used by windows for memory allocation are as follows:
* CoTaskMemAlloc
* GlobalAlloc
* HeapAlloc etc.
* **CoTaskMemAlloc:**
* The initial contents of the returned memory block are undefined i.e., there is no guarantee that the block has been initialized, so you should initialize it in your code.
* The allocated block may be larger than *cb* bytes because of the space required for alignment and for maintenance information.
* The syntax of CoTaskMemAlloc function is:

LPVOID CoTaskMemAlloc(

SIZE\_T cb

);

* If *cb* is 0, CoTaskMemAlloc allocates a zero-length item and returns a valid pointer to that item. If there is insufficient memory available, CoTaskMemAlloc returns NULL.
* Memory allocated by CoTaskMemAlloc is freed using **CoTaskMemFree**.

void CoTaskMemFree(

\_Frees\_ptr\_opt\_ LPVOID pv

);

**Parameters:**

pv: A pointer to the memory block to be freed. If this parameter is **NULL**, the function has no effect.

* **CoTaskMemRealloc** changes the size of previously allocated block of task memory.
* Syntax for CoTaskMemRealloc is

LPVOID CoTaskMemRealloc(

LPVOID pv,

SIZE\_T cb

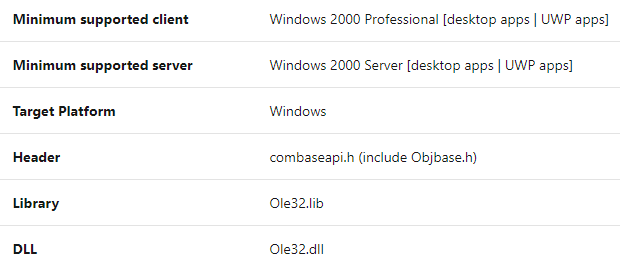
);

**Parameters:**

pv: A pointer to the memory block to be freed. If this parameter is **NULL**, the function has no effect.

cb:The size of the memory block to be reallocated, in bytes. This parameter can be 0.

* **Requirements:**



* **GlobalAlloc:**
* The GlobalAlloc function allocates a selector that could be used to access the amount of memory requested.
* In 16-bit Windows, memory is accessed through values called “selectors”, each of which could address up to 64K.
* If the GlobalAlloc function succeeds, it allocates at least the amount of memory requested. If the actual amount allocated is greater than the amount requested, the process can use the entire amount. If the heap does not contain sufficient free space to satisfy the request, GlobalAlloc returns NULL.
* The syntax for GlobalAlloc is:

DECLSPEC\_ALLOCATOR HGLOBAL GlobalAlloc(

UINT uFlags,

SIZE\_T dwBytes

);

**Parameters:**

dwBytes: The number of bytes to allocate.

* Memory allocated with this function is guaranteed to be aligned on an 8-byte boundary.
* To free the memory, **GlobalFree** function is used.
* The syntax for GlobalFree is:

HGLOBAL GlobalFree(

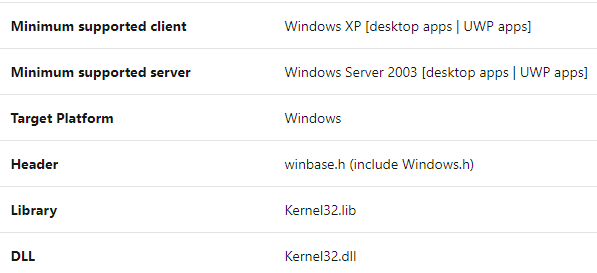
\_Frees\_ptr\_opt\_ HGLOBAL hMem

);

**Parameters:**

hMem: A handle to the global memory object.

* **Requirements:**



* **HeapAlloc:**
* Allocates a block of memory from a heap.
* The allocated memory is not movable.
* The address returned by HeapAlloc is valid until the memory block is freed or reallocated; the memory block does not need to be locked, because the system cannot compact a private heap, it can become fragmented.
* If the HeapAlloc function succeeds, it allocates at least the amount of memory requested.
* Syntax for HeapAlloc is:

DECLSPEC\_ALLOCATOR LPVOID HeapAlloc(

HANDLE hHeap,

DWORD dwFlags,

SIZE\_T dwBytes

);

**Parameters:**

hHeap:A handle to the heap from which the memory will be allocated.

dwBytes: The number of bytes to allocate.

* Memory allocated by HeapAlloc is freed using **HeapFree.**
* Syntax for HeapFree is:

BOOL HeapFree(

HANDLE hHeap,

DWORD dwFlags,

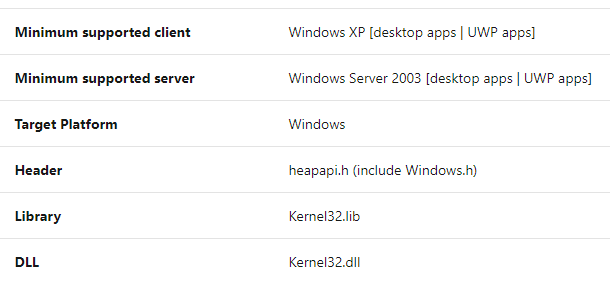
\_Frees\_ptr\_opt\_ LPVOID lpMem

);

**Parameters:**

lpMem:A pointer to the memory block to be freed.

* After that memory is freed, any information that may have been in it is gone forever.
* **Requirements:**



**MacOS X**

* The system library provides **malloc()**, which is the user-space memory allocation function.
* malloc has the following syntax:

void\* malloc(size\_t size);

* malloc is included in the <stdlib.h> header.
* malloc implementation uses an abstraction called malloc zone.
* malloc family contains various other functions for memory allocation:
* **calloc():** to allocate contiguous memory with every element initialized to zero.
* **valloc():** to allocate memory which is aligned on a page boundary.
* **realloc():** to change the size of already allocated memory.
* **reallocf():** if the requested memory is not allocated the pointer passed is freed unlike in realloc().
* The memory is deallocated using the **free()** function.
* The syntax of free function is:

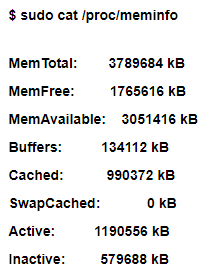
void\* free(void \*ptr);

* Memory allocation APIs in Carbon and Core Foundation(C-based framework) are implemented on top of malloc().
* User programs can use stack based memory allocator **alloca()**. This allows allocation of temporary space in runtime stack.
* The allocated memory this way is freed during a subsequent invocation of the function.

**PHYSICAL MEMORY MANAGEMENT**

**LINUX**

* When Linux uses system RAM, it creates a virtual memory layer to then assign processes to virtual memory.
* This extra abstraction layer is here so that each running process doesn't overlap and try to use memory already being used by another process. This also means that virtual memory can be expanded beyond the physical RAM capacity, which can be useful in a pinch even if it's not very efficient.
* Any file or part of a file system is mapped using the system command mmap, and is referred to as a memory mapped file.
* Sometimes it happens that a process is occupying memory that's needed for another one. In this case, the OS uses the OOM (out of memory) killer. This utility chooses a process and reallocates its memory pages to other processes. In Linux, this OOM killer is enabled by default.
* "Cgroups" is a utility used to isolate a process to a specific memory address, which groups processes into logical groups and allocates an amount of memory to them.
* Linux is capable of working with multiprocessor systems using NUMA (non-uniform memory access). NUMA allocates memory to processes running on the CPU closest to the physical RAM.
* To view memory information on Linux:



**WINDOWS**

* Physical memory refers to actual RAM chips or modules, typically installed on a computer’s motherboard.
* **Hardware Reserved (296 MB)** is the memory space reserved by hardware drivers which must be always remain on the RAM. This memory is essentially locked and is not available to the memory manager.
* **In Use (1648 MB)** memory is the sum total of Working Sets of all running processes owned by the operating system, kernel (non-paged pool), drivers and the various applications.
* **Modified (52 MB)** memory contains modified pages that have been removed from process working sets, because it was idle for long. Modified memory contents must be written to disk before it can be repurposed.
* **Standby (1111 MB):** When a process exits normally, the memory manager moves the unmodified pages in the working set to the Standby memory, which effectively makes the Standby memory a true cache of recently used files.
* **The Free (989 MB)** memory are the locations that have not yet been allocated to any process OR were previously allocated but returned to the memory manager for reuse when the process ended.
* When a process requests for a page, the memory manager first looks for the page in the Standby memory and if available, returns it as a working set. This is called repurposing a page. If the page is not present in Standby, the memory manager loads it from the Hard Disk into the Free Memory which then becomes part of the process working set.
* The memory manager maintains a thread that wakes up periodically to initialize pages on the Free page list so that it can move them to the Zero page list. The Zero page list contains pages that have been initialized to zero, ready for use when the memory manager needs a new page.

**MacOS**

* The primary concern of the original engineers appears to have been fragmentation.
* To solve this, Apple engineers used the concept of a relocatable handle, a reference to memory which allowed the actual data referred to be moved without invalidating the handle. Apple's scheme was simple - a handle was simply a pointer into a (non relocatable) table of further pointers, which in turn pointed to the data.
* If a memory request required compaction of memory, this was done and the table, called the master pointer block, was updated.
* Originally the Macintosh had 128 kB of RAM, with a limit of 512 kB. This was increased to 4 MB upon the introduction of the Macintosh Plus. These Macintosh computers used the 68000 CPU, a 32-bit processor, but only had 24 physical address lines. The 24 lines allowed the processor to address up to 16 MB of memory (224 bytes).
* This was fixed by changing the memory map with the Macintosh II and the Macintosh Portable, allowing up to 8 MB of RAM.