Memory Management

Aakash Badiyani, Jeet Kapadia,

Karthik Jain & Mohit Samarth

May 11, 2016

CSC 540 Advance Operating System

California Lutheran University

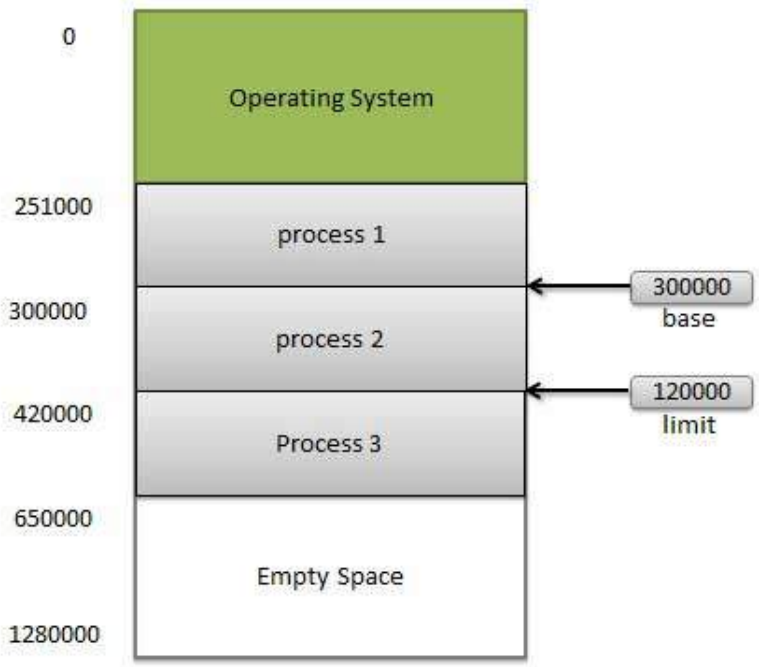
Table of Contents

Abstract

Memory management is an important part of an operating system which handles or manages primary memory. Memory management keeps track of each and every memory location. It chooses which process will get memory at what time. It tracks at whatever point some memory gets liberated or unallocated and correspondingly it upgrades the status. There are various concepts, which are included in memory management one of which is Dynamic loading. It is a mechanism where the program is loaded into main memory only when it is called. The other Concepts which is included in this would be Dynamic Linking, Swapping, Virtual Memory, Paging, Page table, Fragmentation, Memory Allocation, Segmentation, Segmentation with Paging.

Memory Management

Memory management is one of the most important parts of an operating system. Next to the CPU, it is one of the most important resources in a computer system. It stores information fed into the computer system giving each piece data a unique address so that it can be referenced later sometime. Therefore, it is very important for an operating system to manage the memory or all the data in the memory can be lost or messed up with some other stuff. This research work mainly concentrates on the concepts of memory management used by an operating system. During the course of this paper we will learn about the contiguous and dynamic memory allocation, different memory management algorithms, i.e. paging and segmentation, comparisons between different algorithms used for dynamic memory allocation. We will also learn about virtual memory, the key concept in modern operating systems and an important tool to manage the memory effectively.



Contiguous Memory Allocation

The operating system and the various user processes must reside in the main memory at the same time. Therefore, the different parts of the main memory must be allocated in the most efficient way possible. In the contiguous memory allocation, the memory is divided into 2 parts. The resident operating system takes up one part and the other is left to the

various user processes. The operating system is usually placed in the low memory due to the presence of the interrupt vector in the low memory. The idea here is to provide each process with its own single contiguous section of memory.

Dynamic Memory Allocation

There can be at a given instant one, i.e. single program environment, or more than one, i.e. multiple program environment, process present in the main memory so the OS has to manage the memory for both the situations. All the programs fed into the memory are made up of modules that require memory for its code or data components. The management here is done at the module level, i.e. linking, dynamic loading and overlays. Linking essentially consists of combining a number of modules to make a single executable process. The linker is generally outside the OS and applies takes care of the constraints applied by the OS.

Dynamic loading is the loading of the module at run time. All routines are kept on the disk in a relocatable format. The OS is responsible for this. It also has to keep track of the number of programs using the module, and is responsible for resolving the references made. The OS also removes the module from the memory when it no longer needed. The advantage of the dynamic loading is that an unused routine is never loaded. This method is particularly useful when large

amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.

Overlays are multiple modules that are sharing the same address when the program is loaded. The OS needs to know which is loaded into an overlay area (an area of storage which must be large enough to hold the biggest module going into it), and to automatically load a different module when it may be needed. In some cases programs manage these overlays, but this is rare as OS handling leads to more standardization of the job. The OS keeps track of all these modules through different addressing modes, for example, absolute (the full memory location), relative (address specified relative to where the program is currently running) and indexed (offset from a specified absolute memory location).

Multiple Program Memory Management

The Having multiple programs in the memory requires partitioning, i.e. dividing the memory into several portions. This is called partitioning.

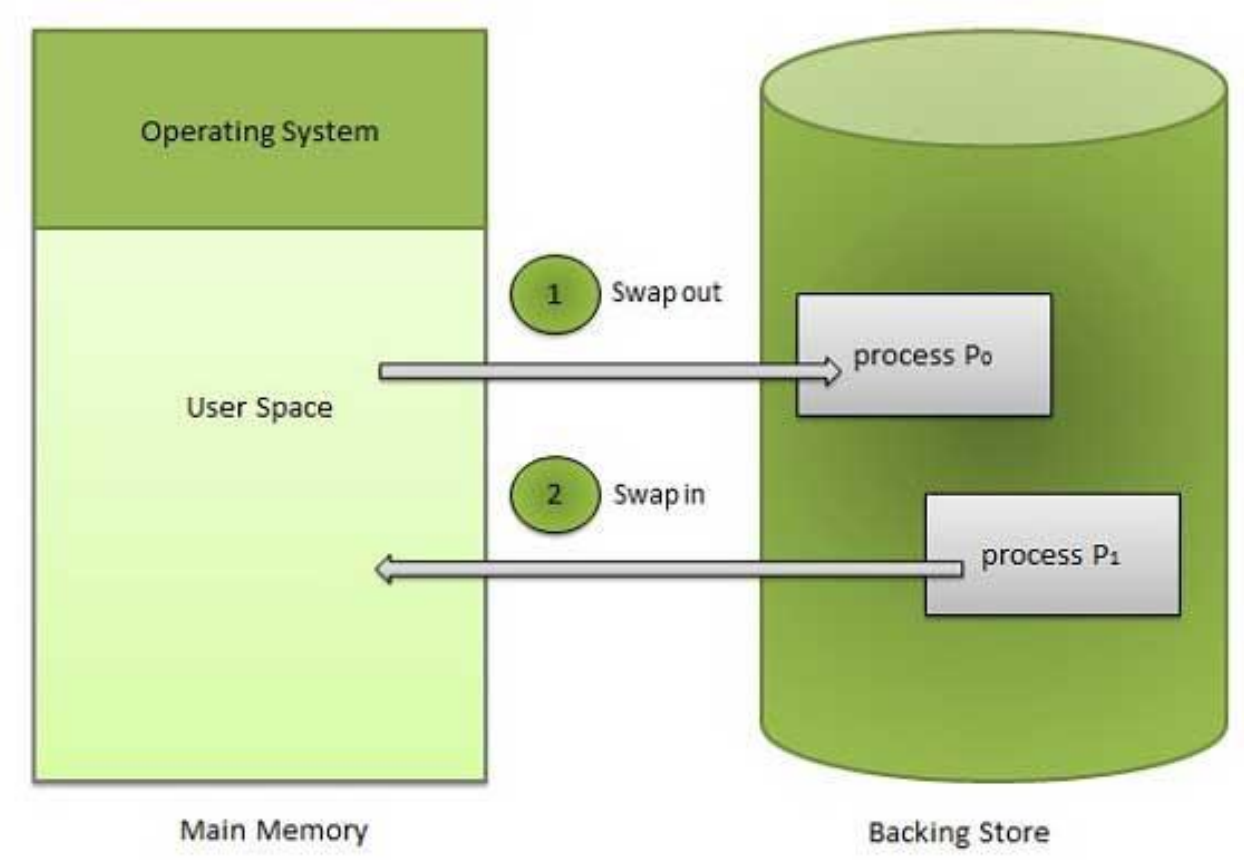
Fixed sized partitions: The main memory is divided into a number of fixed-sized partitions. Only one process can reside in a partition. This is called the multiple-partition method. Whenever a partition is free, a process from the input queue is selected and 6 loaded into the partition. Upon the termination of the process, the partition becomes free for another process.

Variable sized partitions: The main memory is divided into portions large enough to fit in the data. The size of the partition can be changed during reallocation. Over-Allocation: Here the main memory is over-allocated. The programs given exceed the memory available, and then some of the

non-running programs are stored into the disk. The program is then moved into the main memory to run and non-running program is moved into the memory. This is called swapping.

Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory to a backing store, and then brought back into memory for continued execution. Backing store is a usually a hard disk drive or any other secondary storage which fast in access and large enough to accommodate copies of all memory images for all users. It must be capable of providing direct access to these memory images. Major time consuming part of swapping is transfer time. Total transfer time is directly proportional to the amount of memory swapped.



Managing Free Memory

Free memory need to be handled in order to have efficient allocation of time and space. In order to perform Free memory management Bitmaps and Lined List methods are employed.

Bitmaps

When memory is assigned dynamically, the operating system must manage it. With a bitmap, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa).

Linked List

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes.

Virtual Memory

A virtual memory system uses disk storage to provide applications with an address space larger than available physical memory. This helps the system execute multiple processes with large address spaces simultaneously. The disk area used by the virtual memory system is called swap space. The virtual memory system uses swap space to store memory pages that are not expected to be of immediate use. Typically, systems tend to remove pages that have not been accessed recently or that are not accessed frequently from memory and store them on disk (called page-out). When a page stored on disk is accessed again, it is brought back into physical memory (called page-in). The page-out/pagein process is transparent to applications (except for performance

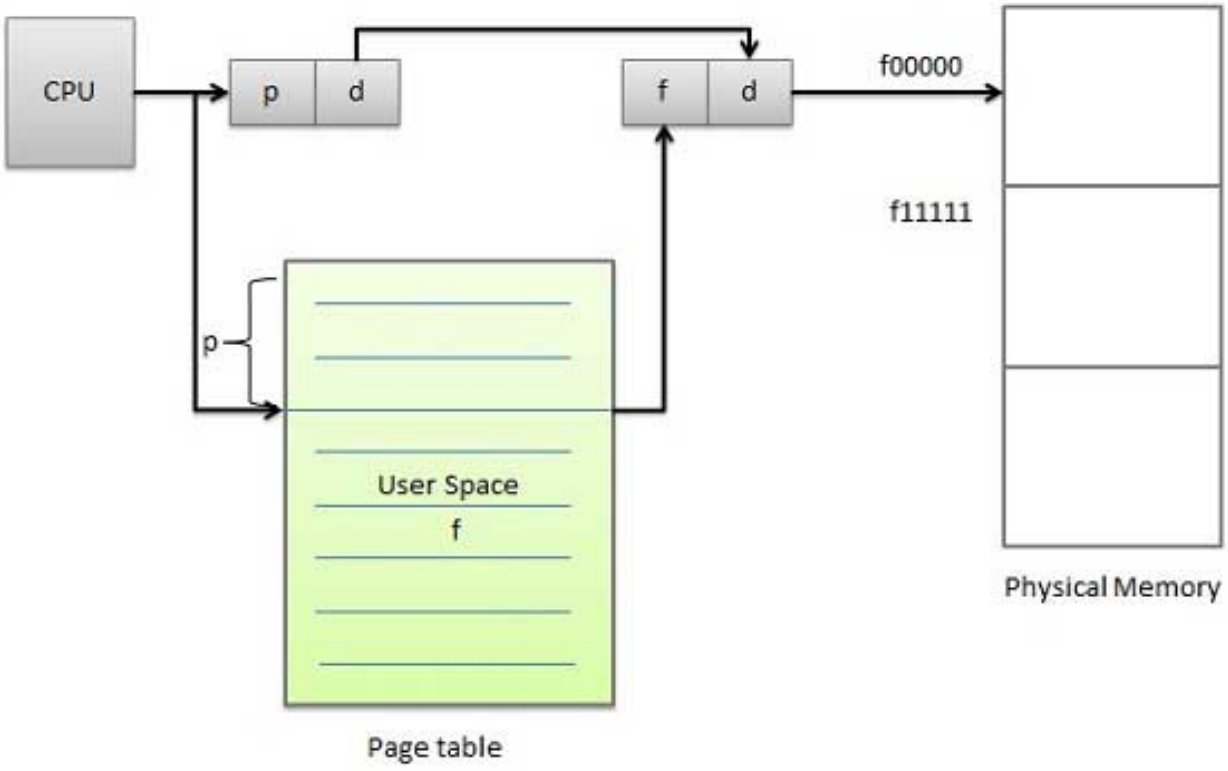
effects). Thus, the virtual memory system is responsible for handling disk errors and maintaining the illusion that the page is actually in physical memory.

Memory Management Unit

A **memory management unit** (**MMU**), sometimes called paged **memory management unit** (PMMU), is a computer hardware unit having all memory references passed through itself, primarily performing the translation of virtual memory addresses to physical addresses.

Paging

External fragmentation is avoided by using paging technique. Paging is a technique in which physical memory is broken into blocks of the same size called pages. When a process is to be executed, it's corresponding pages are loaded into any available memory frames. Logical address space of a process can be non-contiguous and a process is allocated physical memory



whenever the free memory frame is available. Operating system keeps track of all free frames. Operating system needs n free frames to run a program of size n pages.

Page Fault Processing

A page fault occurs when a process accesses a virtual page for which there is no PTE in the page table or whose PTE in some way prohibits the access, e.g., because the page is not present or because the access is in conflict with the access rights of the page. Page faults are triggered by the CPU and handled in the page_fault_handler.

Page Table

It is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are used by the accessing process, while physical addresses are used by the hardware, or more specifically, by the RAM subsystem.

Multi-level Page Tables

The basic idea behind a multi-level page table is simple. First, chop up the page table into page-sized units; then, if an entire page of page-table entries (PTEs) is invalid, don't allocate that page of the page table at all. To track whether a page of the page table is valid (and if valid, where it is in memory), use a new structure, called the page directory. The page directory thus either can be used to tell you where a page of the page table is, or that the entire page of the page table contains no valid pages.

Inverted Page Table

An even more extreme space savings in the world of page tables is found with inverted page tables. Here, instead of having many page tables (one per process of the system), we keep a single page table that has an entry for each physical page of the system. The entry tells us which process is using this page, and which virtual page of that process maps to this physical page.

Virtualized page table

It was mentioned that creating a page table structure that contained mappings for every virtual page in the virtual address space could end up being wasteful. But, we can get around the excessive space concerns by putting the page table in virtual memory, and letting the virtual memory system manage the memory for the page table.

Nested page table

Nested page tables can be implemented to increase the performance of hardware virtualization. By providing hardware support for page-table virtualization, the need to emulate is greatly reduced. For x86 virtualization the current choices are Intel's Extended Page Table feature and AMD's Rapid Virtualization Indexing feature.

Dynamic Linking

Linking is the process of collecting and combining various modules of code and data into a executable file that can be loaded into memory and executed. Operating system can link system

level libraries to a program. When it combines the libraries at load time, the linking is called static linking and when this linking is done at the time of execution, it is called as dynamic linking.

In static linking, libraries linked at compile time, so program code size becomes bigger whereas in dynamic linking libraries linked at execution time so program code size remains smaller.

Segmentation

Even though someone can access objects in the process through a two dimensional address, the actual physical memory is still only one-dimensional. This brings the need for a segment table, which will allow the 2-D address to be mapped to a 1-D physical address. The segment table is actually nothing more than a simple array of base-limit register pairs. A segment address consists of a segment number, s, and an offset into that segment, d. The segment number is used to index your way into the segment table. The offset is added to the segment to produce the address (physical) of the desired memory. This offset must fall between 0 and the segment limit. If this rule is violated the OS will produce an error.

Page Replacement Algorithm

FIFO (First-in, first-out): This algorithm keeps the page frames in an ordinary queue, moves a frame to the tail of the queue when it loaded with a new page, and always chooses the frame at the head of the queue for replacement, i.e. uses the frame whose page has been in memory the longest. While this algorithm may seem at first glance to be reasonable, it is actually about as bad as you can get. The problem is that a page that has been memory for a long time could equally likely be frequently used or unused, but FIFO treats them the same way.

RAND (Random): This algorithm simply picks a random frame. This algorithm is also pretty bad.

OPT (Optimum): This one picks the frame whose page will not be used for the longest time in the future. If there is a page in memory that will never be used again, its frame is obviously the best choice for replacement. Otherwise, if (for example) page A will be next referenced 8 million instructions in the future and page B will be referenced 6 million instructions in the future, choose page A. This algorithm is sometimes called Belady's MIN algorithm after its inventor. It can be shown that OPT is the best possible algorithm and gives the smallest number of page faults. Unfortunately, OPT, like SJF processor scheduling, is implementable because it requires knowledge of the future. Its only use is as a theoretical limit.

LRU (Least Recently Used): This algorithm picks the frame whose page has not been referenced for the longest time. The idea behind this algorithm is that page references are not random. Processes tend to have a few pages that they reference over and over again. A page that has been recently referenced is likely to be referenced again in the near future. LRU is actually quite a good algorithm. There are two ways of finding the least recently used page frame. One is to maintain a list. Every time a page is referenced, it is moved to the head of the list. When a page fault occurs, the least-recently used frame is the one at the tail of the list. Unfortunately, this approach requires a list operation on every single memory reference, and even though it is a pretty simple list operation, doing it on every reference is completely out of the question, even if it were done in hardware. An alternative approach is to maintain a counter or timer, and on every reference store the counter into a table entry associated with the referenced frame. On a page fault, search

through the table for the smallest entry. This approach requires a search through the whole table on each page fault, but since page faults are expected to tens of thousands of times less

frequent than memory references, that's ok. Unfortunately, all of these techniques require hardware support and nobody makes hardware that supports them. Thus LRU, in its pure form, is just about as impractical as OPT.

NRU (Not Recently Used): There is a form of support that is almost universally provided by the hardware: Each page table entry has a referenced bit that is set to 1 by the hardware whenever the entry is used in a translation. The hardware never clears this bit to zero, but the OS software can clear it whenever it wants. With NRU, the OS arranges for periodic timer interrupts and on each “tick”, it goes through the page table and clears all the referenced bits. On a page fault, the OS prefers frames whose referenced bits are still clear, since they contain pages that have not been referenced since the last timer interrupt. The problem with this technique is that the granularity is too coarse. If the last timer interrupt was recent, all the bits will be clear and there will be no information to distinguished frames from each other.

SLRU (Sampled LRU): This algorithm is similar to NRU, but before the referenced bit for a frame is cleared it is saved in a counter associated with the frame and maintained in software by the OS. One approach is to add the bit to the counter. The frame with the lowest counter value will be the one that was referenced in the smallest number of recent “ticks”. This variant is called NFU (Not Frequently Used). A better approach is to shift the bit into the counter (from the left). The frame that hasn't been reference for the largest number of “ticks” will be associated with the counter that has the largest number of leading zeros. Thus we can approximate the least-recently used frame by selecting the frame corresponding to the smallest value (in binary). This only approximates LRU for two reasons: It only records whether a page was referenced during a tick, not when in the tick it was referenced, and it only remembers the most recent n ticks, where n is

the number of bits in the counter. We can get as close an approximation to true LRU, as we like, at the cost of increasing the overhead, by making the ticks short and the counters very long.

Second Chance: When a page fault occurs, this algorithm looks at the page frames one at a time, in order of their physical addresses. If the referenced bit is clear, then it chooses the frame for replacement, and returns. If the referenced bit is set, give the frame a "second chance" by clearing its referenced bit and going on to the next frame (wrapping around to frame zero at the end of memory). Eventually, a frame with a zero referenced bit must be found, since at worst, the search will return to where it started. Each time this algorithm is called, it starts searching where it last left off. This algorithm is usually called CLOCK because the frames can be visualized as being around the rim of a clock, with the current location indicated by the second hand.

Fixed Allocation: This algorithm gives each process a fixed number of page frames. When a page fault occurs it uses LRU or some approximation to it, but only considers frames that belong to the faulting process. The trouble with this approach is that it is not at all obvious how to decide how many frames to allocate to each process. If you give a process too few frames, it will thrash. If you give it too many, the extra frames are wasted.

Page-Fault Frequency (PFF): This approach is similar to fixed allocation, but the allocations are dynamically adjusted. The OS continuously monitors the fault rate of each process, in page faults per second of virtual time. If the fault rate of a process gets too high, either give it more pages or swap it out. If the fault rate gets too low, take some pages away. When you get back enough pages this way, either start another job (in a batch system) or restart some job that was swapped out. The problem is choosing the right values of "too high" and "too low".

Working Set: The Working Set (WS) algorithm is as follows: Constantly monitor the 'working set' of each process. Whenever a page leaves the working set, immediately take it away

from the process and add its frame to a pool of free frames. When a process page faults, allocate it a frame from the pool of free frames. If the pool becomes empty, we have an overload situation, the sum of the working set sizes of the active processes exceeds the size of physical memory so one of the processes is stopped. The problem is that WS, like SJF or true LRU, is not implementable. A page may leave a process' working set at any time, so the WS algorithm would require the working set to be monitored on every single memory reference. That's not something that can be done by software, and it would be totally impractical to build special hardware to do it. Thus all good multi-process paging algorithms are essentially approximations to WS.

Clock: Some systems use a global CLOCK algorithm, with all frames, regardless of current owner, included in a single clock. As we said above, CLOCK approximates LRU; so global CLOCK approximates global LRU, which, as we said, is not a good algorithm. However, by being a little careful, we can fix the worst failing of global clock. If the clock “hand” is moving too “fast” (i.e., if we have to examine too many frames before finding one to replace on an average call), we can take that as evidence that memory is over-committed and swap out some process.

WSClock: An interesting algorithm has been proposed (but not, to the best of my knowledge widely implemented) that combines some of the best features of WS and CLOCK.

Conclusion

Memory allocation is one of the most important duties of an operating system. In the numerous methods memory allocation dynamically, the best is the first-fit method, since it is quick, and minimizes fragmentation. Virtual memory is a common method used to increase the size of the memory space, by replacing frames in the physical memory with pages from the virtual memory. This benefits the operating system in the sense that a process does not have to be completely in memory to execute. To implement a virtual memory system properly, the algorithm

with as few page faults as possible must be used to minimize page replacement. The least-recently used algorithm was the best for performance, but the enhanced second-chance algorithm uses several ideas of the LRU method, and is easier to implement. The problems of fragmentation, both internal and external can develop, as well as thrashing can occur even with most careful planning. But, the effects of these problems can be minimized with a careful plan, and an effective memory management system can be implemented.

References

[1] Tanenbaum, Andrew S. Modern Operating Systems. 4th ed. Herbert Bos, 2000.

[2] Wesley, Addison. Fundamental Algorithms. 3rd ed. Donald Knuth, 1997.

[3] "Memory Management Method, Memory Storage Device and Memory Controlling Circuit Unit" in Patent Application Approval Process." Information Technology Newsweekly. 15 Mar. 2016.

[4] "Introduction to Memory Management." Web. 23 Mar. 2016. <www.memorymanagement.org/mmref/>