MEMORY MANAGEMENT

Term Paper

Operating Systems

CS-384

Submitted to: Dr. Taylor

Submitted by: Deepak Agrawal

Submitted on: February 2, 2003

Table of contents (Index)

Introduction

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 and see how Windows NT manages its memory making use this concept.

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).

Single Program Memory Management Models:

1. **The Bare Machine model**
2. **The Resident Monitor model**

In the **Bare Machine model**, the OS manages nothing; instead the entire address space is given to the program and all the managing is left to it.

In the **Resident Monitor model**, the spot where the OS is located in the memory is specified, i.e. the high or the low end, and the rest of the memory is given to the program. In today’s windows systems the OS is loaded in the low end, so the program has to be relocatable, because they have to be loaded above the OS, wherever that happens to be at that time. If the OS is loaded in the high end then the program can be loaded into the same low-end spot.

Multiple Program Memory Management

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 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.

Dynamic Memory Allocation Algorithms

An available block of memory to store the process is called a hole. There are 4 algorithms to allocate the memory dynamically:

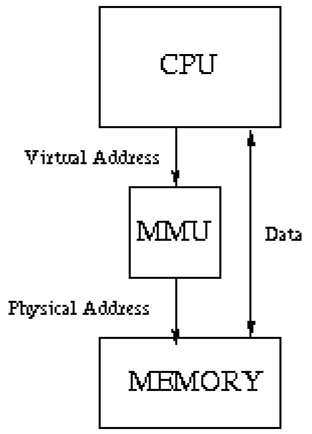
1. **First-fit:** Allocate the first hole that is big enough. Searching start at the beginning of the set of holes we can stop searching as soon as we find a free hole that is large enough.
2. **Next-fit:** This behaves exactly as the first fit except that the scan begins from where the previous one left off.
3. **Best-fit:** Allocate the smallest hole big enough. We must search the entire list, unless the list is kept ordered by size. This strategy produces the smallest leftover hole.
4. **Worst-fit:** Allocate the largest hole. Again we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Dynamic Memory Allocation Efficiency

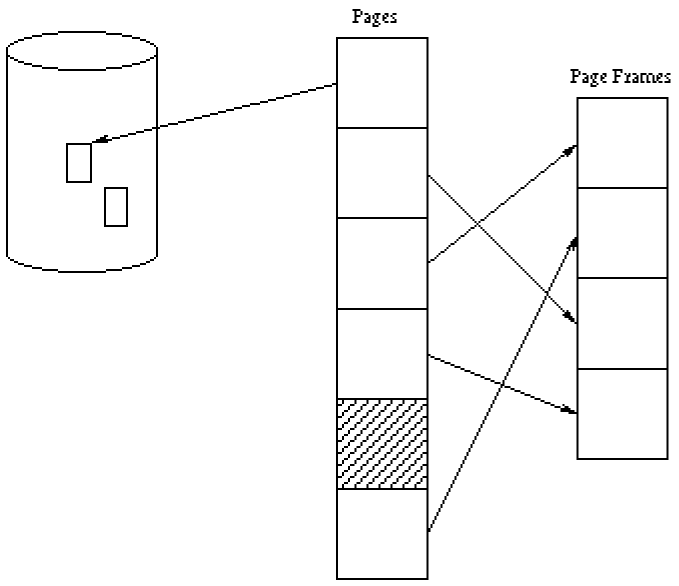
A study done of the efficiency of next-fit, first-fit, and best-fit showed that in some cases, next-fit performs worse than first-fit or best-fit. When the mean size of the block is less than one-sixteenth the available memory, first-fit performs the best of all, with best-fit close to the performance of first-fit, and next-fit being substantially inferior to both of them. After that, the three methods produce very similar results. One hypothesis is that when first-fit outperformed best-fit, it was due to first-fit filling one end of memory first, leaving large blocks at the other end, which would be sufficient for larger allocations. First-fit and best-fit also outperform worst-fit, in both time to allocate, and efficient use of memory.

Paging

This is a memory scheme that permits the physical-address space to be noncontiguous. Most modern computers have special hardware called a memory management unit (MMU). This unit sits between the CPU and the memory unit. Whenever the CPU wants to access memory (whether it is to load an instruction or load or store data), it sends the desired memory address to the MMU, which translates it to another address before passing it on the memory unit. The address generated by the CPU, after any indexing or other addressing-mode arithmetic, is called a virtual address, and the address it gets translated to by the MMU is called a physical address.



Each page is a power of 2 bytes long, usually between 1024 and 8192 bytes. In other words, each page is mapped to a contiguous region of physical memory called a page frame.



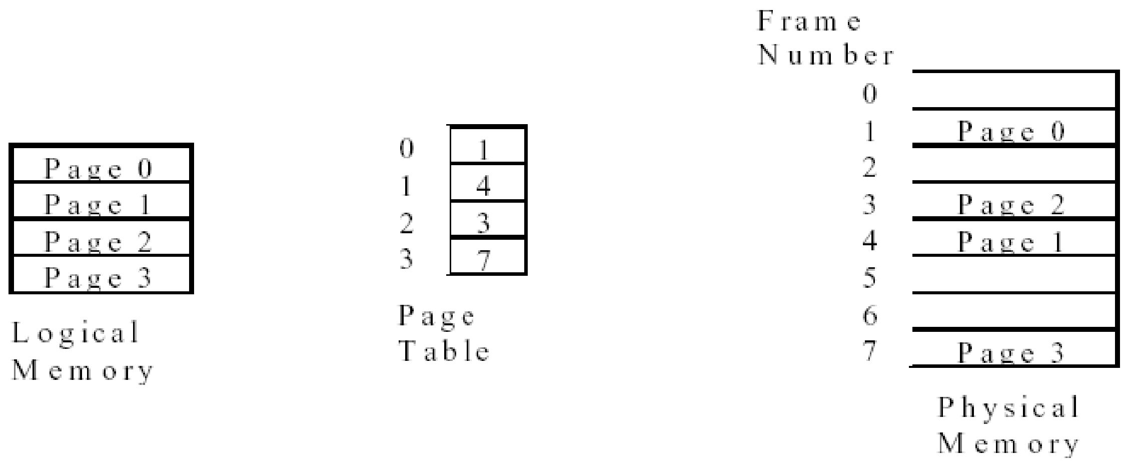
The MMU allows a contiguous region of virtual memory to be mapped to page frames scattered around physical memory making it easier for the OS when allocating memory. Much more importantly, it also allows pages, not stored frequently to be stored on disk. The tables used by the MMU have a valid bit for each page in the virtual address space. If this bit is set, the translation of virtual addresses on a page proceeds as normal. If it is clear then any attempt to access an address on the page results in the generation of an interrupt called page fault trap. The OS has an interrupt handler for page faults. It is the job of this handler to get the requested page into memory. When a page fault is generated for page the interrupt handler does the following:

- Find out where the contents of page are stored on disk. The OS keeps this information in a table. If the page isn't anywhere at all, the OS takes some corrective action such as killing the process that made the reference.

Assuming the page is on disk:

- Find another page mapped to some frame of physical memory that is not used much.
- Copy the contents of frame out to disk.
- Clear the page's valid bit so that any subsequent references to page will cause a page fault.
- Copy the initial page's data from disk to frame.
- Update the MMU's tables so that the initial page is mapped to frame.
- Return from the interrupt, allowing the CPU to retry the instruction that caused the interrupt.

To implement a paging system, the physical memory is divided into multiple frames. The logical memory (process) is divided into multiple pages. When one page of the process is needed, it is loaded into a frame in the physical memory. Remember that the pages and frames are of the same size, so this system will not produce any wasted space between the frames or pages. The operating system maintains a page table, which holds the base address of each page in the physical memory. The logical address generated by the CPU consists of a page number and a page offset. The page number is used to look up the base address of that page in the page table. The page offset is the offset into the page, starting at the base address.



Paging System Implementation

Each page-table entry contains a ‘valid’ bit as well as some other bits. These other bits include:

Protection

At a minimum one bit to flag the page as read-only or read/write. Sometimes more bits to indicate whether the page may be executed as instructions, etc.

Modified

This bit, usually called the *dirty bit*, is set whenever the page is referenced by a write (store) operation.

Referenced

This bit is set whenever the page is referenced for any reason, whether load or store.

Page Replacement

All of the hardware methods for implementing paging have one thing in common, whenever the CPU generates a virtual address for which the corresponding page table entry is marked **invalid**, the MMU generates a page fault interrupt and the OS must handle the fault. There are three possible reasons for the OS to mark the page as invalid:

- There is a bug in the program being run. In this case the OS simply kills the program.
- Unix treats a reference just beyond the end of a process' stack as a request to grow the stack. In this case, the OS allocates a page frame, clears it to zeros, and updates the MMU's page tables so that the requested page number points to the allocated frame.
- The requested page is on disk but not in memory. In this case, the OS allocates a page frame, copies the page from disk into the frame, and updates the MMU's page tables so that the requested page number points to the allocated frame.

In all but the first case, the OS is faced with the problem of choosing a frame. If there are any unused frames, the choice is easy, but that is not the frequent case. If the memory is being heavily used, the choice of frame becomes crucial for good performance. We will first consider page-replacement algorithms for a single process, and then consider algorithms to use when there are multiple processes.

Page Replacement Algorithms

Frame Allocation for a Single Process

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 unimplementable 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.

Frame Allocation for Multiple Processes

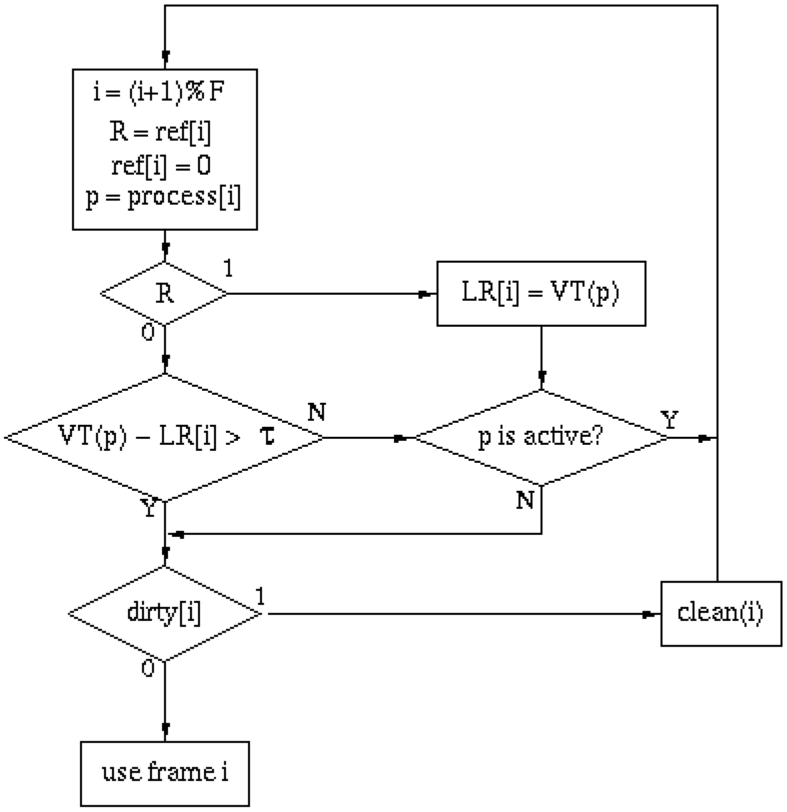
Fixed Allocation: This algorithm gives each process a fixed number of page frames. When a page fault occurs it uses LRU or some approximation to it, but only considers frames that belong to the faulting process. The trouble with this approach is that it is not at all obvious how to decide how many frames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

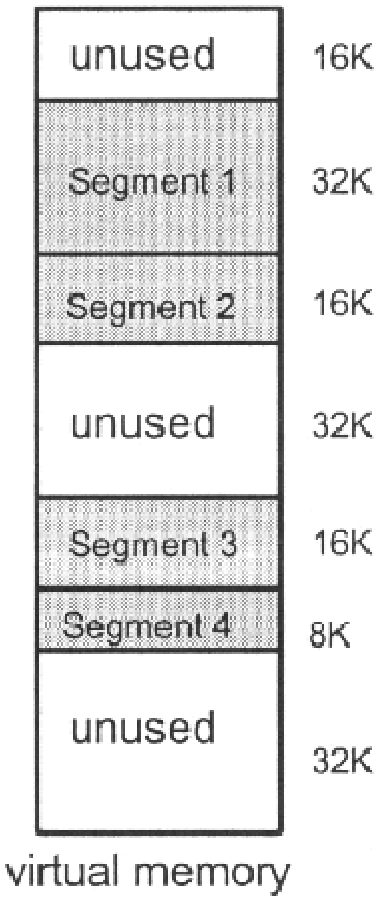
WSClock: An interesting algorithm has been proposed (but not, to the best of my knowledge widely implemented) that combines some of the best features of WS and CLOCK. Assume that we keep track of the current virtual time *VT(p)* of each process *p*. Also assume that in addition to the reference and dirty bits maintained by the hardware for each page frame *i*, we also keep track of *process[i]* (the identity of process that owns the page currently occupying the frame) and *LR[i]* (an approximation to the time of the last reference to the frame). The time stamp *LR[i]* is expressed as the last reference time according to the virtual time of the process that owns the frame.



In this flow chart, the WS parameter (the size of the window in virtual time used to determine whether a page is in the working set) is denoted by the Greek letter *tau*. The parameter *F* is the number of *frames*, i.e. the size of physical memory divided by the page size. Like CLOCK, WSClock walks through the frames in order, looking for a good candidate for replacement, cleaning the reference bits as it goes. If the frame has been referenced since it was last inspected, it is given a ``second chance". (The counter *LR[i]* is also updated to indicate that page has been referenced recently in terms of the virtual time of its owner.) If not, the page is given a ``third chance" by seeing whether it appears to be in the working set of its owner. The time since its last reference is approximately calculated by subtracting *LR[i]* from the current (virtual) time. If the result is less than the parameter *tau*, the frame is passed over. If the page fails this test, it is either used immediately or scheduled for cleaning (writing its contents out to disk and clearing the dirty bit) depending on whether it is clean or dirty. There is one final complication: If a frame is about to be passed over because it was referenced recently, the algorithm checks whether the owning process is active, and takes the frame anyhow if not. This extra check allows the algorithm to grab the pages of processes that have been stopped by the load-control algorithm. Without it, pages of stopped processes would never get any ``older'' because the virtual time of a stopped process stops advancing. Like CLOCK, WSClock has to be careful to avoid an infinite loop. As in the CLOCK algorithm, it may a complete circuit of the clock finding only dirty candidate pages. In that case, it has to wait for one of the cleaning requests to finish. It may also find that all pages are unreferenced but "new". In either case, memory is over-committed and some process needs to be stopped.

Segmentation

In segmentation implementation the virtual address space is divided into a collection of segments of varying length. Each of these segments has a name and a length (offset) associated with it. The segment name and the offset within it are specified by an address. For a user to specify an address the user must provide two things: a segment name and an offset. The advantage of segmentation over paging is that it uses less amount of hardware.



Segmenting Virtual Memory

Fragmentation

Fragmentation is the state in memory where there is free memory, enough to satisfy a request, but it is spread out in multiple small blocks, so there is no contiguous block of memory large enough to satisfy the request. There are two types of fragmentation.

1. **External fragmentation**, where the unused blocks of memory exist outside of a process. This problem is present in the dynamic memory allocation algorithms.
2. **Internal fragmentation**, where the memory has been allocated to a process, but remains unused by that process. Paging systems suffer from this type of fragmentation.

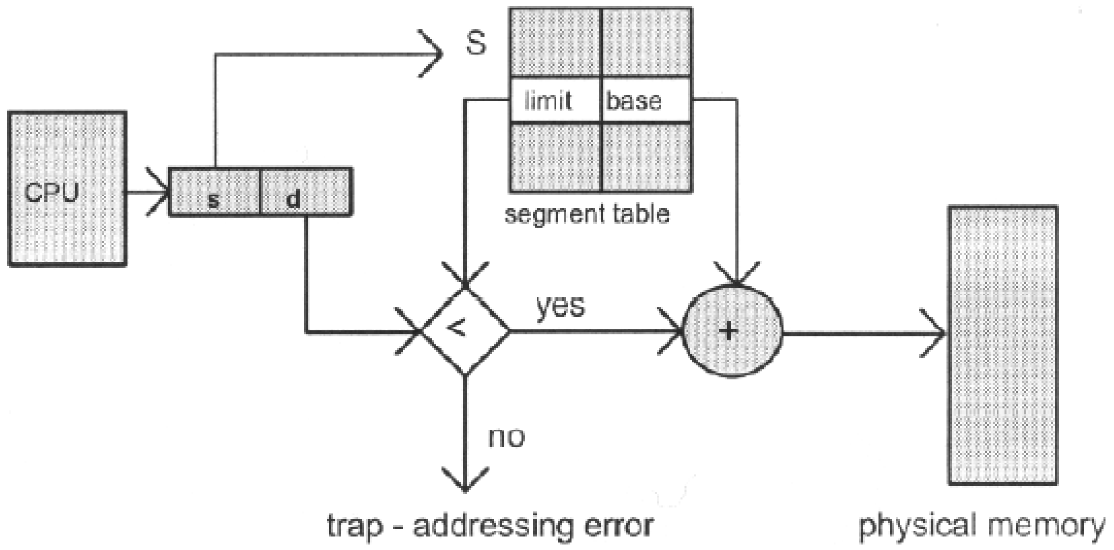
Thrashing

During thrashing the operating system spends all of its time replacing frames from memory, and does not execute the processes. This condition can be caused by a common series of events. Thrashing can be avoided by denying a thrashing application the opportunity to take frames from another. If a thrashing process were allowed to take frames from another process, that process would soon begin thrashing also. Another way to stop thrashing is to limit the number of processes that can be executed at any one.

Hardware Support

Segmentation

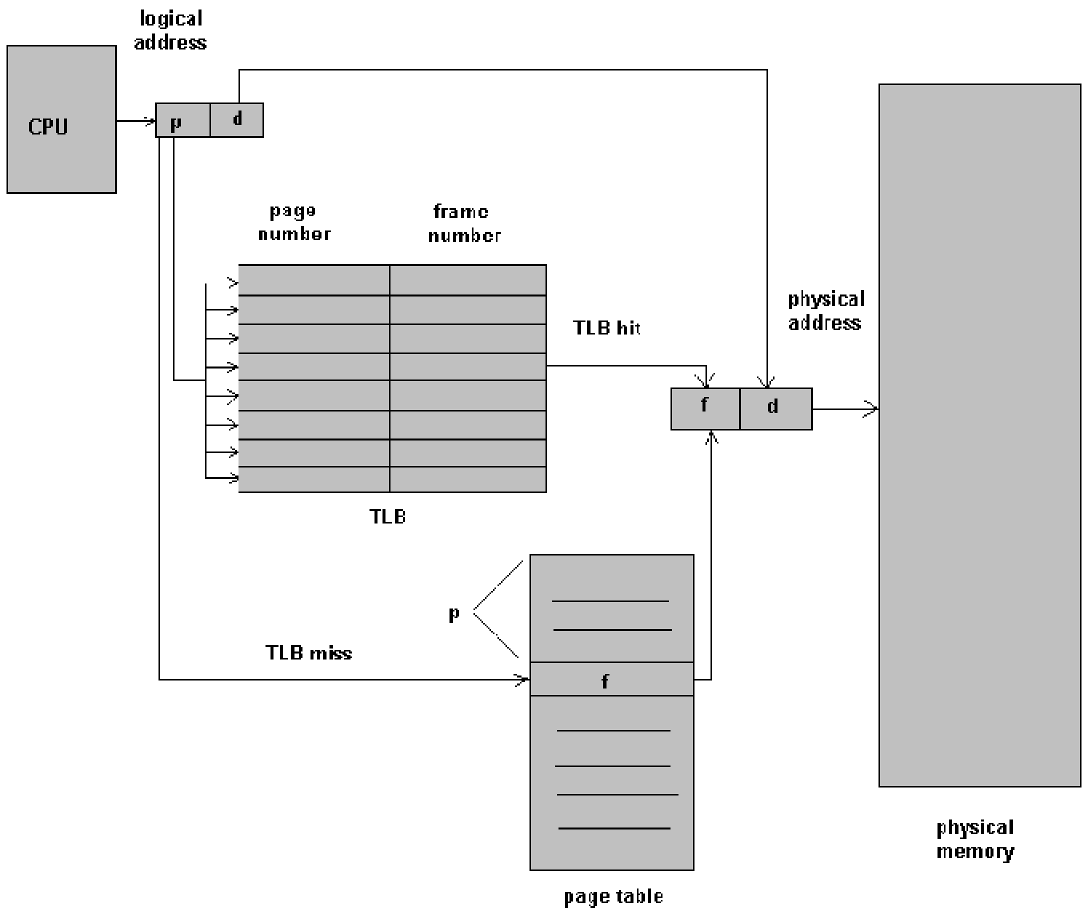
Even though someone can access objects in the process through a two dimensional address, the actual physical memory is still only one-dimensional. This brings the need for a segment table, which will allow the 2-D address to be mapped to a 1-D physical address. The segment table is actually nothing more 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.



Segmentation Hardware

Paging

The paging systems require a **translation look-aside buffer (TLB)** in order to solve the problem of having to access memory 2 times to access a byte of information. The TLB is a small fast-lookup hardware cache or in other words an associative, high-speed memory. Each entry in the TLB consists of 2 parts: a key (or tag) and a value. When the associative memory is presented with an item, it is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. Only a few of the page table entries are contained in the TLB. When CPU generates a logical address it's compared to the ones in the TLB. If it is found, the frame number is available immediately and is used to access memory. In case of a **TLB miss,** a memory reference to the page is made and the page number and the frame number are added in the TLB. The OS replaces this one with some other entry in case the TLB is full. The TLB entries for kernel code are often **wired down**, meaning that they cannot be removed form the TLB.



Paging Hardware with TLB

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.

Bibliography

Galvin, Peter B., and Abraham Silberschatz Operating System Concepts, 4th Edition. 1995

Kaiser, Stephen H. (*Stephen Hendrick*) The Design of Operating System for small computer systems

The Seventh IEEE Workshop on Future Trends of Distributed Computing Systems, Dec 1999

IEEE Transactions on Software Engineering, Jan. 1980, “Working Sets Past and Present”
Denning, Peter J.

Bays, Carter 1977. “A Comparison of Next-fit, First-fit, and Best-fit” *Communications of the ACM*, Volume 20, Number 3, March 1977

Belady, L.A.; R.A. Nelson; and G.S. Shedler. 1969. “An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine.” C*ommunications of the ACM*, Volume 12, Number 6, June 1969

Operating system Tutorials
http://www.themoebius.org.uk/tutes/memory1.html