Virtual Memory

Arnav Gupta

March 26, 2024

Contents

1 Hardware and Control Structures 2
1.1 Virtual Memory . . . 2
1.2 Locality and Virtual Memory . . . 3
1.3 Paging . . . 3
1.3.1 Page Table Structure . . . 4
1.3.2 Inverted Page Table . . . 4
1.3.3 Translation Lookaside Buffer . . . 5
1.3.4 Page Size . . . 6
1.4 Segmentation . . . 7
1.4.1 Virtual Memory Implications . . . 7
1.4.2 Organization . . . 7
1.5 Combined Paging and Segmentation . . . 8
1.6 Protection and Sharing . . . 8

2 Operating System Software 9
2.1 OS Policies for Virtual Memory . . . 9
2.2 Fetch Policy . . . 10
2.3 Placement Policy . . . 10
2.4 Replacement Policy . . . 11
2.4.1 Frame Locking . . . 11
2.4.2 Basic Algorithms . . . 11
2.4.3 Page Buffering . . . 13
2.4.4 Replacement Policy and Cache Size . . . 14
2.5 Resident Set Management . . . 14
2.5.1 Resident Set Size . . . 14
2.5.2 Replacement Scope . . . 15
2.6 Cleaning Policy . . . 17

2.7 Load Control 17
2.7.1 Multiprogramming Level 17
2.7.2 Process Suspension 18

1 Hardware and Control Structures

Key characteristics of paging and segmentation are:

1. All memory references within a process are logical addresses that are dynamically translated into physical adddresses at runtime, allowing a process to be swapped in and out of main memory (can occupy different regions of main memory at different times)
2. A process can be broken into pieces (pages or segments) that need not be contiguous in main memory during execution

This ultimately means that not all pages or segments of a process need to be in main memory during execution, only the next instruction to be fetched and next data location to be accessed must be in main memory.

As a process executes, the procesor can determine if a memory reference is in the resident set using the page/segment table, if not it can generate an interrupt for a memory access fault. This blocks the process so that the OS can bring that piece of the process into main memory using a disk IO read request.

1.1 Virtual Memory

Virtual memory: a storage allocation scheme in which secondary memory can be addressed as though it were part of main memory, where addresses a program uses are different from addresses the memory system uses to identify storage sites through translation

Amount of virtual storage is limited by the addressing scheme of the computer and the amount of secondary memory available, not main memory.

Virtual address: address assigned to a location in virtual memory to allow that location to be accessed as if it were in main memory

Virtual address space: virtual storage assigned to a process

Address space range of memory addresses available to a process

Real address: address of a storage location in main memory

Resident set: portion of a process that is in main memory

Implications of virtual memory:

1. More processes maintainted in main memory since only portions of each process are in main memory, which more efficiently uses the processor especially with context switching.
2. Processes can be larger than main memory so programmers do not need to worry about the size of their program as the OS manages virtual memory based on paging or segmentation.

1.2 Locality and Virtual Memory

Over a short period of time, execution could be confined to a small section of a program and so loading the full program in would be wasteful in case it is suspended. Saves time as unused pieces are not swapped in and out of memory.

Thrashing: when the system spends most of its time swapping pieces rather than executing instructions because pieces are thrown out just before they are used

Locality: program and data references within a process tend to cluster, used to avoid thrashing

Locality suggests virtual memory may be effective, but this requires:

- hardware support for paging and/or segmentation
- OS includes software for manageing movement of pages and/or segments between secondary and main memory

1.3 Paging

With virtual memory paging, there are equal size pages of the same length as frames but not all pages need to be loaded into main memory frames for execution.

Virtual memory paging maintains a page table for each process, but page table entries must have:

- a present (P) bit to indicate whether or not they are in main memory (if so the frame number is correct)

- a modify (M) bit to indicate whether the contents of the corresponding page have been altered since the page was last loaded into main memory (if so, write the page out on replacement)
- other control bits for protection, sharing, etc.

1.3.1 Page Table Structure

Reading a word from memory involves the translation of a logical (virtual) address consisting of page number and offset into a physical address with frame number and offset using the page table.

Page table must be in main memory to be accessed. When a process is running, a register holds the starting address of the page table for that process. Page number of a virtual address is used to index the table and find the frame number which is combined with the offset of the virtual address to find the real address.

Number of pages in a process can exceed the number of frames in main memory.

If processes are allowed to have large amounts of virtual memory, memory devoted to page tables alone could be unacceptably high. Most virtual memory schemes store page tables in virtual memory, so page tables are subject to paging.

When a process is running, at least the page table entry of the currently executing page must be in memory (can be more).

Some processors use a 2-level scheme to organize large page tables:

- there is a page directory where each entry points to a page table
- the number of pages allowed for a process is the product of the number of entries in the page directory and the max number of entries in a page table
- max length of a page table is typically at most one page

1.3.2 Inverted Page Table

Page tables so far have size proportional to virtual address space.

Inverted page tables can be used where:

- page number of a virtual address is mapped to a hash value

- hash value is a pointer to the inverted page table which contains page table entries
- one entry in the inverted page table for each real memory page frame rather than one per virtual page, so page table has fixed size
- chaining may be needed because of hash

Each entry in the page table includes:

- **page number**: page number portion of virtual address
- **process identifier**: process that owns the page, where this and page number identifies a page within the virtual address space of a process
- **control bits**: includes flags like valid, referenced, modified, and protection and locking info
- **chain pointer**: null field if no chained entries, otherwise field contains index value of next entry in chain

1.3.3 Translation Lookaside Buffer

Every virtual memory reference is 2 physical memory accesses, one for the page table entry and one for the desired data. Regular virtual memory scheme would then double memory access time.

To overcome this, a special high-speed cache for page table entries called a **translation lookaside buffer** (TLB) is used. Contains the most recently used page table entries.

The procedure with a TLB is:

- processor first checks TLB for desired page table entry
  - if hit, frame number is retrieved and real address is formed
  - if miss, then processor uses page number to index process in page table and examine the corresponding page table entry
- if the P bit is set then the page is in main memory and the processor can get the frame number from the page table entry to form the real address
  - this also updates the TLB to include the page table entry

- if the P bit is not set, a page fault occurs and the OS is invoked to load the page and update the page table

Since the TLB only contains some entries in a full page table, each entry in the TLB must include the page number and the complete page table entry. Hardware allows a number of TLB entries to be checked simultaneously to check if there is a page number match, called **associative mapping**.

TLB design relies on hardware cache design and this must consider the way in which entries are organized and replacement policy.

Virtual memory must interact with main memory cache. Once the real address is generated, it is in the form of a tag and the remainder, which the cache uses to check if the block containing that word is present. If so, it is returned, and if not, the word is retrieved from main memory.

1.3.4 Page Size

Factors to consider:

- **internal fragmentation**: smaller the pages, less internal fragmentation
- **pages per process**: smaller the pages, more pages per process and larger page tables
  - can result in double page fault to bring portion of page table, and then process page
- **physical characteristics**: most secondary-memory are rotational and favour a larger page size for more efficient block transfer of data

If page size is very small, many pages will be in memory near recent references keeping page fault rate low. As page size increases, each individual page contains locations further from recent references, so less locality and page fault rate rises. As the page size approaches the size of the process, page fault rate falls.

Page fault rate is also determined by the number of frames allocated to a process, and this falls as the number of pages in main memory grows.

As main memory grows, address space used by programs does as well. Programming techniques can avoid locality, such as:

- OOP which encourages using many small programs data modules scattered over many objects
- multithreaded applications which can have abrupt changes in instructions and scattered memory references

For given TLB size, as memory size of processes grows and locality decreases, hit ratio of TLB accesses decreases, so TLB can be a bottleneck. Larger TLB can help or larger page sizes so that each page table entry refers to a larger block of memory.

Multiple page sizes can be used which provides flexibility in using a TLB effectively. Most OSs only support one page size.

1.4 Segmentation

1.4.1 Virtual Memory Implications

Segmentation allows memory to be viewed as consisting of agments of unequal, dynamic size. Memory references consist of a segment number and offset.

Advantages to the programmer:

1. simplifies handling growing data structures as they can be assigned a segment whose size changes dynamically and can be swapped out of main memory if required
2. allows programs to be altered and recompiled independently without requiring all to be relinked and reloaded using multiple segments
3. allows sharing among processes as a utility or useful data can be placed in a segment that can be referenced by other processes
4. allows protection as segments can be constructed to contain a well-defined set of programs or data that the program or admin can assign privileges

1.4.2 Organization

A unique segment table is associated with each process.

Segment table entries include:

- a bit to indicate if the segment is in memory (if so, starting address and length are valid)

- a modify bit to indicate whether the contents have been altered since the segment was loaded into main memory (if so, weite the segment out to disk)
- other control bits for protection and sharing

Segment table must be in main memory to be accessed. A register holds the starting address of the segment table for the running process.

1.5 Combined Paging and Segmentation

Paging is transparent to the programmer, eliminates external framentation (efficient memory use), can develop sophisticated memory management algorithms to exploit the behaviour of programs because of equal size.

Segmentation is visible to the programmer, handles growing data structures, modularity, and support for sharing and prtection.

Some processors and OS use both for both benefits.

In combined paging/segmentation, address space is broken into segments (by the programmer) that are broken into fixed-size pages. If a segment is smaller than a page, it occupies one page.

Logical address (for system) contains a segment number, page number, and offset. Each process has a segment table and many page tables (one for each process segment).

Procedure:

- with the virtual address, the processor uses the segment number to index into the process segment table and find the page table
- then the page number portion is used to index the page table and look up the corresponding frame number
- this is combined with offset to give a real address

1.6 Protection and Sharing

Segmentation allows for protection and sharing policies.

Since each segment table entry includes length and base address, a program cannot access a main memory location beyond the segment limits.

For sharing, a segment can be referenced in the segment tables of multiple processes.

Same mechanisms available for paging, but specification is difficult since pages not visible to programmer.

A ring-protection structure can be used where inner rings enjoy greater privilege. A program can only access data on the same or outer rings. A program can call services on the same or inner rings.

2 Operating System Software

OS memory management design depends on:

1. whether or not to use virtual memory
2. use of paging, segmentation, or both
3. algorithms used for various aspects of memory management

Virtual memory and paging/segmentation depend on available hardware. Outside of older PC OSs, all provide virtual memory and pure segmentation is rate.

Key issue is performance by minimizing page fault rate to reduce overhead.

While process switching during page fetch, would like to arrange so that when process is executing, probability of finding a missing page is minimized.

2.1 OS Policies for Virtual Memory

- **Fetch policy**: demand paging or prepaging
- **Placement policy**
- **Replacement policy**: basic are optimal, LRU, FIFO, clock, and non-basic is page buffering
- **Resident Set Management**: resident set size is either fixed or variable, replacement scope is either global or locality
- **Cleaning Policy**: demand or precleaning
- **Load Control**: degree of multiprogramming

Performance of any set of policies depends on:

- main memory size
- relative speed of main and secondary emmory
- size and number of processes competing for resources
- execution behaviour of individual programs, depends on nature of applications, programming languages, compiler, style of programmer, and user behaviour

2.2 Fetch Policy

Determines when a page should be brought into main memory.

Demand paging: a page is brought into main memory only when a reference is made to a locations on that page

This causes:

- many page faults on process start
- as more pages on main memory, page faults drop to a very low level

Prepaging: pages other than the one demanded by a page fault are brought in, such as contiguous pages

This is ineffective is most extra pages are not referenced. Can be employed when a process started (programmer defines desired pages) or when a page fault occurs (invisible to programmer).

2.3 Placement Policy

Determines where in real memory a process piece is to reside.

Already discussed for pure segmentation, but for pure paging or paging/segmentation, placement is usually irrelevant since address translation hardware and main memory access hardware can perform functions for any page-frame combo with equal efficiency.

Important for NUMA (nonuniform memory access) multiprocessor as memory can be referenced by any processor on the machine but acess time for a physical location varies with distance between processor and memory. For NUMA, automatic placement strategy is desirable to assign pages to memory that provide the best performance.

2.4 Replacement Policy

Deals with the selection of a page in main memory to be replaced when a new page is brought in.

Related to:

- number of page frames allocated per active process
- if set of pages for replacement should be limited to the process that caused the page fault or encompass all page frames in main memory
- among set of pages considered, which page should be selected for replacement

First 2 are resident set management, 3rd is replacement policy.

The more elaborate and sophisticated the replacement policy, the greater the hardware and software overhead to implement it.

2.4.1 Frame Locking

Some frames can be locked, where the page stored in that frame cannot be replaced.

Examples include the kernel, key control structures, IO buffers, and time critical areas that are locked into main memory frames.

Achieved by associating a lock bit with each frame that is in the frame table or in the current page table.

2.4.2 Basic Algorithms

Optimal selects for replacing the page for which the time to the next reference is the longest, though impossible to implement and just a judge against real algorithms

Least Recently Used (LRU) replaces the page in memory that has not been referenced by the longest time.

- works by locality since that page is least likely to be used again in the near future
- does nearly as well as optimaly, but difficult to implement
  - could tag each page with time of last reference but too much overhead

    - could maintain a stack of page references, but expensive

First-In-First-Out (FIFO) treats page frames as a circular buffer and pages are removed for a process in round-robin style

- only requires a pointer that circles through page frames of a process
- simple to implement
- does not work well if regions of program are heavily used throughout life of a program

Clock policy requires association of an additional bit with each frame (use bit)

- when a page is loaded into a frame in memory and on each reference, use bit is set to 1
- for page replacement, the set of frames that are candidates for replacement is a circular buffer with an associated pointer
    - when a page is replaced, pointer indicates next frame in the buffer after the last one updated
    - when replacing a page, OS scans a buffer to find a frame with use bit 0
    - when a frame with use bit 1 is found, it is reset to 0
- similar to FIFO except that any frame with use bit 1 is passed over
- can be made more powerful by increasing the number of bits it employs such as a modify bit associated with each page in main memory
    - works by ensuring a page is not replaced until it has been written back into secondary memory
- with the use and modify bits, each frame falls into:
    - not accessed recently, not modified
    - accessed recently, not modified
    - not accessed recently, modified
    - accessed recently, modified
- then the clock algorithm works as:

  - begin at the current position of the pointer, scan the frame buffer and do not change the use bit, first frame with use and modify both 0 is selected for replacement
  - if this is not possible, look for a frame with use bit 0 and modify bit 1, but on this scan set the use bit to 0 on each frame bypassed
  - if this is not possible, the pointer returns to the original position and all frames in the set have use bit 0, so go back to 1st step

- this new algorithm is better since is avoids writing back if not needed and by locality, finds a page that will likely not be needed again soon

Other than optimal, order of least page fault rate is LRU, clock, FIFO. Small factor differences can have a noticeable effect on main memory requirements (to avoid degrading OS performance) or OS performance (to avoid enlarging main memory).

2.4.3 Page Buffering

Cost of replacing a page that has been modified is greater than for one that has not, due to write back.

Page buffering: uses FIFO for replacement but to improve performance the replaced page is assigned to the free page list if it has not been modified or the modified page list if so

- page not physically moved in main memory but entry in page table is placed in the corresponding list on replace

Free page list: for page frames available for reading in pages, where a small number of frames are free at all times

- when a page is read in, the page frame at the head of the free page list is used, and this destroys the page there
- the unmodified page remains in memory and its page frame is added to the tail of the free page list

When a modified page is to be written out and replaced, its page frame is added to the tail of the modified page list.

The page to be replaced remains in memory, so if the page is referenced again it can be returned to the resident set at low cost.

Since modified pages are written out in clusters, this significantly reduces the number of IO operations and the amount of disk access time.

2.4.4 Replacement Policy and Cache Size

With larger caches, the replacement of virtual memory pages can have a performance impact as cache blocks for pages are lost on replacement as well.

With page buffering, it is possible to improve cache performance by using a page placement policy in the page buffer (rather than page replacement).

Essence of these strategies is to bring consecutive pages into main memory in a way to minimize the number of page frames mapped into the same cache slots.

2.5 Resident Set Management

2.5.1 Resident Set Size

With paged virtual memory, not all pages of a process need to be in main memory. OS must decide how many pages to bring in.

Factors to decide this:

- less memory per process means more processes in main memory, so higher probability that OS will find at least one ready process so less time lost to swapping
- if a small number of pages of a process are in main memory, more page fualts
- beyond a certain size, more main memory allocation for a process has no noticeable effect on page fault rate for that process

Fixed-allocation Policy: gives a process a fixed number of frames in main memory within which to execute

- number decided at process creation and depends on type of process or programmer
- when a page fault occurs, one page of that process must be replaced by the needed page

Variable-allocation Policy: number of page frames allocated to a process is varied over process lifetime

- if many page faults, then more page frames and vice versa
- relates to the concept of replacement scope
- requires more software overhead

2.5.2 Replacement Scope

Both policies activated by a page fault when there are no free page frames.

Local replacement policy: chooses only among resident pages of a process that generated the page fault in selecting a page to replace

Global replacement policy: considers all unlocked pages in main memory as candidates for replacement, regardless of ownership

No difference in performance, but local easier to analyze and global easier to implement.

Fixed-resident set implies local replacement, otherwise not fixed size.

Variable-allocation can be either local or global.

1. Fixed Allocation, Local Scope A process running in main memory has a fixed number of frames and on a page fault the page to be replaced is from the currently resident pages for the process.

   Amount of allocation given to process is decided based on type of application and amount requested by the program.

   If too little allocated, high page fault rate, if too much allocated, too few programs in main memory so too much swapping.

2. Variable Allocation, Global Scope Easiest to implement.

   OS maintains a list of free frames, and when a page fault occurs, a free frame is added to the resident set of a process and the page is brought in.

   A process experiencing page faults grows in size which reduces page faults.

   Replacement selection is made from all frames in memory, so the process that suffers the reduction is resident set size may not be optimum.

   To counter potential performance problems, use page buffering so that the choice of which page to replace is less significant as the page can be reclaimed if referenced before the next block of pages are overwritten.

3. Variable Allocation, Local Scope Overcomes problems of global-scope.

   Strategy is:

   (a) when a new process is loaded into main memory, allocate to it a certain number of page frames as its resident set based on application type, program request or other criteria, with prepaging or demand paging to fill the allocation

   (b) when a page fault occurs, select the page to replace from among the resident set of the process that suffers the fault

   (c) from time to time, reevaluate the allocation provided to the process and increase/decrease it to improve overall performance

   With this, changes to resident set size are deliberate and based on the likely future demands of active processes.

   More complex but better performance. Key elements are criteria used to determine resident set size and timing of changes.

   Working set strategy: set of pages of a process that have been referenced in the last t logical time units

   For many programs, periods of stable working set sizes alternate with periods of rapid change. Processes begin transient, then go back and forth between stability and transition.

   Strategy for resident set size:

   (a) monitor the working set of each process

   (b) periodically remove from the resident set of a process pages that are not in its working set, basically LRU

   (c) a process may execute only if its working set is in main memory

   This exploits locality to minimize page faults.

   Problems with working set strategy:

   (a) past does not always predict future, so size and membership of working set will change over time

   (b) true measurement of working set for each process is impractical, as this would require timestamps and time-ordered queue of pages

(c) optimal value of t is unknown and would vary

Real implementations monitor page fault rate rather than working set directly. Page fault rate low means smaller resident set, and vice versa.

2.6 Cleaning Policy

Concerned with determining when a modified page should be written out to secondary memory.

Demand cleaning: a page is written to secondary memory only when selected for replacement.

- writing a dirty page is coupled with reading a new one
- minimized page writes but a process with a page fault must wait for 2 page transfers before becoming unblocked

Precleaning: writes modified pages before their page frames are needed so pages can be written out in batched

- allows the writing of pages in batches, but this only makes sense when many pages have been modified before they are replaced
- otherwise wastes transfer capacity of secondary memory

Better approach uses page buffering by cleaning only pages that are replaceable, but cleaning and replacement are decoupled. Replaced pages can be placed on modified and unmodified lists, where modified list can periodically be written out in batches and moved to unmodified list. A page on the unmodified list is either reclaimed if referenced or lost when frame assigned to another page.

2.7 Load Control

Concerned with determining the number of processes resident in main memory.

Too few processes resident means all blocked is more frequent, too many processes resident means many page faults, so thrashing.

2.7.1 Multiprogramming Level

Number of processes resident in main memory.

Working set or page-fault frequency monitoring implicitly incorporate load control.

$L = S$ criterion adjusts the multiprogramming level so the mean time between faults is the mean time required to process a page fault, which yields maximum processor utilization.

50% criterion attempts to keep utilization of the paging device at approx. 50%, which also yields maximum processor utilization.

Could also adapt clock page replacement algorithm by monitoring the rate at which the pointer scans the circular buffer of frames. If the rate is below a threshold, either few page faults are occurring or for each request, average number of frames scanned is small so many resident pages are not referenced. In both cases, multiprogramming level can be increased. High pointer scan rate implies opposite.

2.7.2 Process Suspension

If the degree of multiprogramming is reduced, resident processes must be suspended.

Possibilities for this are:

- **lowest-priority process**: implements scheduling policy decision and unrelated to performance issues
- **faulting process**: greater probability that faulting task does not have its working set resident so performance would suffer least by suspending it and blocks a process that will be blocked anyways
- **last process activated**: process least likely to have its working set resident
- **process with smallest resident set**: least future effort to reload, but penalizes programs with strong locality
- **largest process**: most free frames
- **process with largest remaining execution window**: approximates a shortest-processing-time-first scheduling discipline