Justin Ciocoi

Dec. 17, 2023

CSCI 375 Textbook Notes

Chapter 10: Virtual Memory

- Virtual memory is a technique by which we can allow the execution of processes that are not completely in memory
- One large advantage of this scheme is that programs can be larger than physical memory

10.1: Background

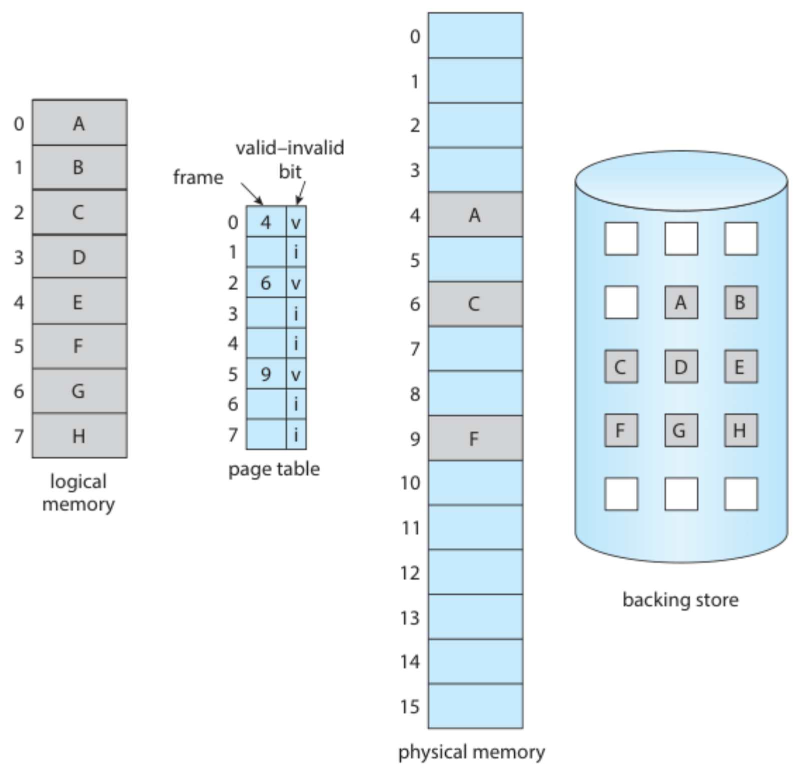
- The memory management algorithms which are outlined in chapter 9 are necessary because of the basic requirement that the instructions being executed must be in physical memory
- The first approach to meeting this requirement is to place the entire logical address space in physical memory
  - Dynamic linking can help to ease this restriction, but it generally requires special precautions and extra work by a programmer
- This requirement seems necessary and reasonable, but it unfortunately limits the size of a program to the size of physical memory in a system
- However, in many cases, the entire program is not needed, such as in the following cases
  - Programs that have code meant to handle unusual error conditions which will seldom, if ever, occur
  - Arrays, lists, and tables are often allocated more memory than they actually need
  - Certain options and features of a program may be used rarely
- Running a program not entirely in physical memory would allow users not only to have a far more efficient multiprocessing environment, but it also allows programmers to design programs for an extremely large virtual address space rather than a limited physical space
- *Virtual memory* involves the separation of logical memory as perceived by developers from physical memory

- This separation then allows for an extremely large virtual memory to be provided to application programmers even if only a limited subset of physical memory is available
- The *virtual address space* of a process refers to the logical view of how a process is stored in memory
    - Typically, this view is that a process begins at a certain logical address, and exists in contiguous memory
- However, this is not always the case as programs sometimes reside in non-contiguous memory
- For any process, we allow its stack and heap to grow towards each other
    - The holes in between the two are useful since they can be used when one of the two wants to grow such as when a library is dynamically linked
- In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two ore more processes through page sharing, leading to the following benefits
    - System libraries such as the standard C library can be shared by several processes through mapping of the shared object into a virtual address space
    - Similarly, memory can be shared, which allows processes to create regions of shared memory where each process has its own virtual copy, but they share the actual physical copy

10.2: Demand Paging

- How is an executable program loaded from secondary storage into memory?
- One option is to load the entire program into physical memory at program execution time
- However, we might not initially need the entire program to be in physical memory
- An alternative strategy is to load pages only as they are needed, which is a technique known as *demand paging*
- With demand-paged virtual memory, pages are loaded only when they are demanded during program execution
- **10.2.1: Basic Concepts**

- The general concept behind demand paging is to load a page in memory only when it is needed
- As a result, throughout the duration of a process's execution, some pages will be in memory, and some will be in secondary storage
  - We therefore need some form of hardware support to distinguish between the two
  - We can use the valid-invalid bit scheme to accomplish this goal



- When the bit is set to valid, the associated page is both legal and in memory, but if the bit is set to invalid, then the page is either not valid, or is valid but is currently in secondary storage
- What happens if a process tries to access a page that was not brought into memory?
  - Access to a page which is marked invalid causes a *page fault*
- The procedure for handling this page fault is fairly straightforward
  1. We check an internal table, which is usually kept with the process control block, in order to determine whether the reference was a valid or invalid memory access
  2. If the reference was invalid, we terminate the process, and if it was valid but we have not yet brought in the page, we now page it in

3. We find a free frame
4. We schedule a secondary storage operation to read the desired page into the newly allocated frame
5. When the storage read is complete, we modify the internal kept with the process and the page table to indicate that the page is now in memory
6. Now, we restart the instruction that was interrupted and the process can access the page as though it had always been in memory

- In the extreme case, we can start executing a process with no pages in memory
    - The process will immediately incur a page fault on its first instruction, and subsequently fault as necessary until every page that it needs is in memory
- This scheme is known as *pure demand paging*
- Theoretically, some programs could access several new pages of memory with each instruction execution, meaning multiple page faults per instruction are possible
- However, this behavior is exceedingly unlikely since programs tend to have a *locality of reference* which results in reasonable performance from demand paging
- The hardware support for demand paging is the same as hardware support for paging and swapping and is as follows
    - *Page table*, which has the ability to mark an entry invalid through a valid-invalid bit or a special value of protection bits
    - *Secondary memory*, which holds pages that are not present in main memory, and is known as the *swap device* which uses the *swap space* to accomplish this goal
- One crucial requirement for demand paging is the ability to restart any instruction after a page fault
- Because we save the state of the interrupted process when the page fault occurs, we must be able to restart the process in *exactly* the same place and state, except for the fact that the requested page is now in memory and is accessible by the program
- As a worst-case example, let us consider a three-address instruction such as adding the content of A to B and placing the result in C
- The following are the steps to execute the above series of instructions

        1. Fetch and decode the instruction
        2. Fetch A
        3. Fetch B
        4. Add A and B
        5. Store the sum in C

    - Even if a page fault occurs when storing the sum in C, the entire list of operations here will be done again, but the repetition comprises of less than one complete instruction and is necessary only whenever a page fault occurs

- **10.2.2: Free-Frame List**
    - When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory
    - To resolve page faults, most operating systems maintain a *free-frame list*, which is a pool of free frames meant to satisfy such requests
    - Operating systems will usually allocate free frames using a technique known as *zero-fill-on-demand*
    - These frames are "zeroed-out" before being allocated, which erases their previous contents
    - When a system starts, all available memory is placed on the free-frame list and as free frames are requested, the size of the list shrinks

- **10.2.3: Performance of Demand Paging**
    - Demand paging can significantly impact the performance of a computer system, and to see why we will compute the effective access time for demand-paged memory

$$\text{effective access time} = (1-p) * ma + p * \text{page fault time}$$

    - In order to compute the effective access time, we must know how much time it takes to service a page fault, which causes the following sequence to occur

        1. Trap to the operating system
        2. Save the registers and process state
        3. Determine that the interrupt was a page fault

4. Check that the page reference was legal and determine the location of the page in secondary storage
5. Issue a read from the storage to a free frame
    1. Wait in a queue until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer if a page to a free frame
6. While waiting, allocate the CPU core to some other process
7. Receive an interrupt from the storage I/O subsystem that the I/O has completed
8. Save the register and process state for the other process, only if step 6 is executed
9. Determine that the interrupt was from the secondary storage device
10. Correct the page table and other tables to show that the desired page is now in memory
11. Wait for the CPU core to be allocated to this process again
12. Restore the registers, process state, and the new page table, and then resume the interrupted instruction

- These are not all always necessary, but in any case, there are three major task components of the page-fault service time
    1. Service the page-fault interrupt
    2. Read in the page
    3. Restart the process
- By looking at the aforementioned formula, we can see that the effective access time for memory is directly proportional to the rate of page-faults
- Thus, in order to keep the slowdown due to paging at a reasonable level, we must keep the probability of a page fault at $p < 0.0000025$ or less than one in every $399,990$ memory accesses

10.3: Copy-on-Write

- Process creation that uses the `fork()` system call may initially bypass the need for demand paging by using a technique which is similar to page sharing
- Recall that `fork()` works by creating a child process that is a duplicate of its parent
- Considering that many child processes invoke the `exec()` system call immediately after creation, the copying of the parent process's address space may be unnecessary
- In these cases, we can use a technique known as *copy-on-write*, which works by allowing the parent and child processes to initially share the same pages
- These pages are marked such that if either process writes to a shared page, then a copy will be created
    - This means as long as the processes are capable of sharing a single copy of a page, they will

10.4: Page Replacement

- Earlier, we assumed that each page faults at most once, when it is first referenced, but this is not strictly accurate
- Over-allocation of memory manifests itself as follows
    - While a process is executing, a page fault occurs
    - The operating system determined where the desired page is residing on secondary storage, but then finds that there are no free frames on the free-frame list, meaning all memory is in use
- The operating system, at this point, has several options, such as attempting to terminate the process
- However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput by not allocating unnecessary memory
- Most operating systems combine swapping pages with *page replacement*, which will be described in detail throughout the remainder of this section
- **10.4.1: Basic Page Replacement**
    - Page replacement fundamentally takes the following approach
        - If no frame is free, we find one that is not currently being used and free it

        - We can free a frame by writing its contents to swap space and changing the page table to indicate that the page is no longer in memory
        - We can now use the freed frame to hold the page for which the process faulted
    - We can modify the page-fault service routine to include page replacement in the following way
        1. Find the location of the desired page on secondary storage
        2. Find a free frame
            1. If there is a free frame, use it
            2. If there is no free frame, us a page-replacement algorithm to select a *victim frame*
            3. Write the victim frame to secondary storage and change the page and frame tables accordingly
        3. Read the desired page into the newly freed frame and change the page and frame tables accordingly
        4. Continue the process from where the page fault occurred
    - If no frames are free, two page transfers are required, which effectively doubles the page-fault service time and increases the effective access time accordingly
    - We can reduce this overhead by using a *modify bit* or *dirty bit*
    - When this scheme is used, each page or frame has a dirty bit associated with it in the hardware
    - Whenever any byte in the page is written into, the dirty bit is set indicating that the page has been modified
    - When we examine a page for page replacement, we can look at its dirty bit, and if it is set, we know that the page has been modified since it was read in from secondary storage
    - If the dirty bit is not set, however, we know that the page has not been modified and does not need to be written back into secondary storage before being replaced
    - In order to implement demand paging, we have to solve two major problems
        - Develop a *frame-allocation algorithm*

        - Develop a *page-replacement algorithm*

    - In general, we will select a page-replacement algorithm by choosing to use the one that exhibits the lowest page-fault rate, since page faults have an enormous impact on system performance

    - When looking at different page-replacement algorithms, we will use the following *reference string*, or string of memory references

$$7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$$

- **10.4.2: FIFO Page Replacement**

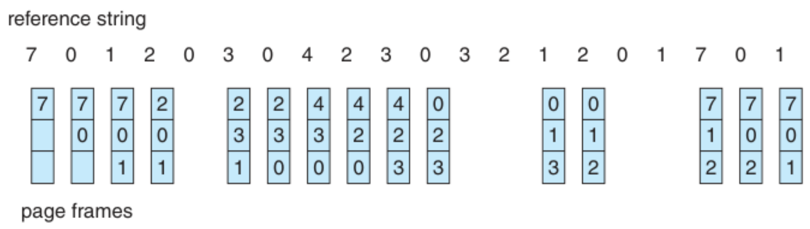


Figure 10.12 FIFO page-replacement algorithm.

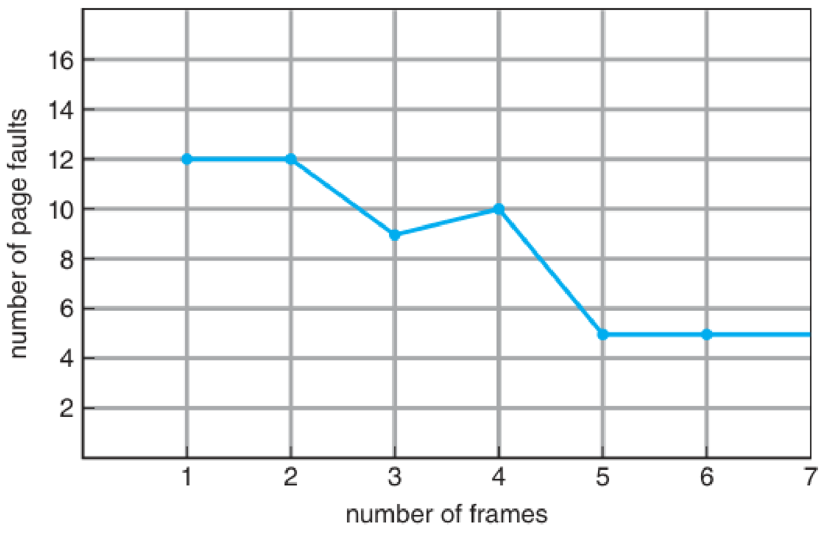
    - This is the simplest page-replacement algorithm and associates with each page the time when that page was brought into memory

    - When a page must be replaced, the oldest page is chosen as the victim page

    - It is not strictly necessary to record the time each page is brought in, as we can instead opt to implement this as a FIFO queue which holds all pages in memory

    - Sometimes, the number of frames used for a page-replacement algorithm does not correlate directly with the rate of page faults

    - This is known as *Belady's Anomaly* and can be seen from the below graph



- **10.4.3: Optimal Page Replacement**

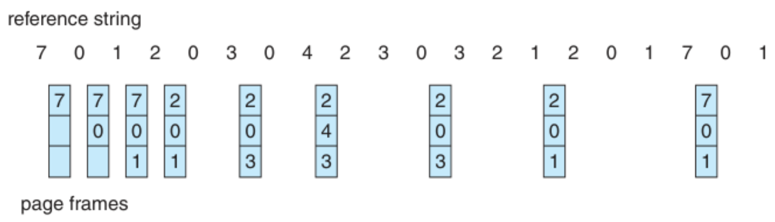


Figure 10.14 Optimal page-replacement algorithm.

  - As a result of the discovery of the above anomaly, the search for an optimal page-replacement algorithm ensued
  - This algorithm is purely theoretical and will opt to replace the page that will not be used for the longest period of time
  - However, because this algorithm requires future knowledge of the reference string, there is no perfect way to implement it, but we can compare other functional algorithms against this one as the gold standard of page-replacement algorithms

- **10.4.4: LRU Page Replacement**

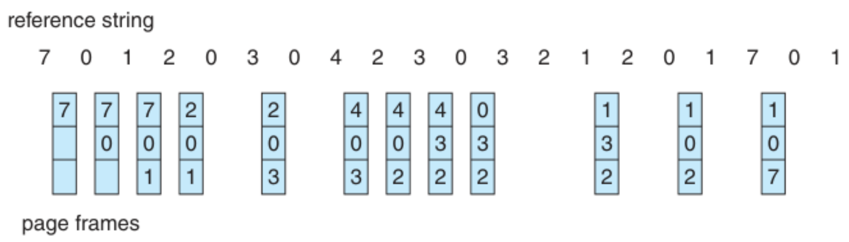


Figure 10.15 LRU page-replacement algorithm.

  - If we use the recent past as an approximation for the future, we can arrive at an algorithm that approximates the optimal one

- We can do this, and then replace the page that has not been used for the longest amount of time
- LRU replacement associates with each page the time of that page's last use
- When a page must be replaced, LRU will choose the page that has not been used for the longest period of time
- This policy is often used and is generally considered to be good, but the general problem is *how* to implement LRU replacement
- Two implementations are feasible
  - *Counters*
    - In the simplest case, we can associate a time-of-use field with each page-table entry and add to the CPU a logical clock or counter, which will be incremented for every memory reference
    - When a reference to a page is made, the contents of the clock register are copied to the time-of-use field
    - In this way, we always have the time of the last reference to each page
  - *Stack*
    - Another approach is to keep a stack of page numbers
    - Whenever a page is referenced, it is removed from the stack and put on the top
    - In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom of the stack
    - Since entries are removed from the bottom of the stack, it is best to implement this stack using a doubly linked list with a head and tail pointer
- LRU does not experience Belady's Anomaly, as it is a *stack algorithm*, meaning it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with $n+1$ frames

- **10.4.5: LRU-Approximation Page Replacement**
  - While the LRU policy seems very advantageous, not many computer systems provide sufficient hardware support for true LRU implementation
  - Hardware support usually comes in the form of reference bits which is set by the hardware whenever a page is referenced
  - Initially, all bits are cleared to 0 by the operating system, and as a process executes, the pit associated with each page is set to 1 by the hardware
  - Thus, we can see which pages have been used, but we do not know the order of use\

- **10.4.5.1: Additional-Reference-Bits Algorithm**
  - Instead of using a single reference bit, we can use a reference byte, consisting of 8 bits
  - When a page is used, the leftmost bit will be set to 1, shifting all other bits right
  - If we interpret these 8 bit bytes as unsigned integers, the page with the lowest number is the LRU page and will be selected by this algorithm
  - This algorithm is also called the *second-chance page-replacement algorithm*
- **10.4.5.2: Second-Chance Algorithm**
  - In essence, this algorithm is a FIFO page-replacement algorithm
  - When a page has been selected, we inspect its reference bit, and if it is set to 1, we give this page a *second chance*
  - A page that is given a second chance has its reference bit cleared and its arrival time updated to the current time
  - Thus, a page given a second chance will not be replaced until all other pages have either been replaced or given a second chance
- **10.4.5.3: Enhanced Second-Chance Algorithm**
  - The second-chance algorithm can be further enhanced through the use of a second reference bit rather than just one
  - This leaves us with 4 possibilities rather than 2
    1. $(0, 0)$: neither used nor modified - best page to replace
    2. $(0, 1)$: not recently used, but modified - not as good, since the page needs to be written out before replacement
    3. $(1, 0)$: recently used but clean - probably will be used again soon
    4. $(1, 1)$: recently used and modified - will probably be used again and needs to be written out to secondary storage before it is replaced
- **10.4.6: Counting-Based Page Replacement**
  - We can also keep a counter to the number of references that have been made to each page, and using these counters develop the following two schemes

        - LFU, or *least frequently used* page replacement will replace the page that has the smallest count
            - However, a page that was heavily used at startup, but wont be for the remainder of a program's life could lead to undesired results when using this algorithm
        - MFU, or *most frequently used* follows a similar idea, but is based on the argument that the page with the smallest count was probably just brought in and has yet to be used

10.5: Allocation of Frames

- How do we allocate the fixed amount of free memory we have in a computer system among various processes
- **10.5.1: Minimum Number of Frames**
    - There exists both a maximum and minimum for frame allocation operations
    - One reason for the existence of a minimum is performance, since a smaller minimum would increase the rate of page-faults
    - The minimum is usually determined by system architecture, whereas the maximum is determined by the total amount of available physical memory
- **10.5.2: Allocation Algorithms**
    - The easiest way to split m frames among n processes is to give each process an equal share of frames
        - This scheme is known as *equal allocation*
    - An alternative is to use *proportional allocation* where a process is allocated a number of free frames proportional to its size as determined by

      $$a_i = \frac{s_i}{S} * m$$

      where $\frac{s_i}{S}$ is equal to the proportion of size taken by this program and m is the number of free frames
- **10.5.3: Global vs. Local Allocation**

    - Another important factor in the way frames are allocated to various processes is page replacement
    - Since multiple processes are competing for frames, we can classify page-replacement algorithms into two categories
        - *Global*
        - *Local*
    - Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process
    - Local replacement requires that each process select only from its own set of allocated frames
    - Consider high- and low-priority processes and how higher priority processes might be more needing of global replacement algorithms whereas lower priority processes are not
    - Global replacement leads to varying process execution time, but generally increased throughput, whereas local replacement might under-utilize memory, but results in more consistent per-process performance
- **10.5.4: Non-Uniform Memory Access**
    - On NUMA systems with multiple CPUs, not all main memory is created equal, and thus a given CPU can access some sections of main memory faster than it can access others
    - Here, optimal performance comes from allocating memory which is architecturally close to the CPU on which the thread is scheduled

10.6: Thrashing

- What might happen if a process does not have the minimum number of frames it needs to support pages in the working set?
- The process will very quickly incur a page-fault, at which point it must select a page to replace
    - However, since all of its pages are in use, it will immediately incur another page-fault, and another, and another
- This repeated paging is called *thrashing* and a process is thrashing when it is spending more time paging than it is executing

- Thrashing results in severe performance problems
- **10.6.1: Cause of Thrashing**
  - Consider the situation where the operating system monitors CPU utilization, and if utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system
  - A global page replacement algorithm is used, and it replaces pages without regard to which process they belong
  - Now suppose a process enters a new phase of execution and needs more frames, at which point it will start faulting and taking pages away from other processes
  - This results in increased paging as those other processes will immediately page fault as well
  - The operating system will now see the CPU utilization decrease as a result of high page-fault rates, and will thus increase the degree of multiprogramming once more, compounding the issue
  - We can limit the effects of thrashing by implementing a *local replacement algorithm*
  - By doing this, we can ensure than if one process starts thrashing, it cannot steal frames from another process and cause that process to thrash as well
  - However, an individual process can still thrash and thus the problem is not yet solved
  - In order to prevent thrashing, we must provide a process with as many frames as it needs, but the issue is knowing how many frames a process needs
  - The *locality model* states that as a process executes it moves from locality to locality, which might overlap
  - If we do not allocate enough frames to a process to accommodate the size of the current locality, the process will thrash since it cannot keep in memory all of the pages which it is actively using
- **10.6.2: Working-Set Model**
  - The *working-set model* is based on the assumption of locality
  - This model uses a parameter, Δ, to define the working set window
  - The general idea is to examine the most recent Δ page references

  - The set of pages in this set is the working set
  - If a page is in use, it will be in the working set and if a page is no longer being used, it will be removed from the working set
  - $WSS_i =$ the total number of pages referenced in the most recent Δ
    - If Δ is to small, it will not encompass the entire locality
    - If Δ is too large, it will encompass several localities
    - If $\Delta = \infty \rightarrow$ will encompass entire program
  - $D = \sum WSS_i =$total demand frames
  - If $D > m$, there is thrashing
  - So, we can implement the policy that if $D > m$, we will suspend one of the processes
  - We can keep track of the working set by using an interval timer as well as a reference bit, but this not completely accurate schemes becomes more and more accurate the more reference bits there are and the shorter the timer interval is
- **10.6.3: Page-Fault Frequency**
  - The working-set model is successful, but it seems like a clumsy way to control thrashing
  - A strategy that uses the page-fault frequency takes a more direct approach
  - If a process has a low page fault rate, it may have too many frames, and if it has a high page fault rate it may not have enough
  - The operating will then make frame allocation decisions based on upper and lower bounds
- **10.6.4: Current Practice**
  - Currently, the best practice in implementing a computer system is to include enough physical memory such that thrashing and swapping is avoided whenever possible