Virtual Memory

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 9

9.1 Background

- Preceding sections talked about how to avoid memory fragmentation by breaking process memory requirements down into smaller bites (pages), and storing the pages non-contiguously in memory. However the entire process still had to be stored in memory somewhere.
- In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:
  1. Error handling code is not needed unless that specific error occurs, some of which are quite rare.
  2. Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
  3. Certain features of certain programs are rarely used, such as the routine to balance the federal budget. :-)
- The ability to load only the portions of processes that were actually needed (and only *when* they were needed) has several benefits:
  - Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer.
  - Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
  - Less I/O is needed for swapping processes in and out of RAM, speeding things up.
- Figure 9.1 shows the general layout of ***virtual memory***, which can be much larger than physical memory:

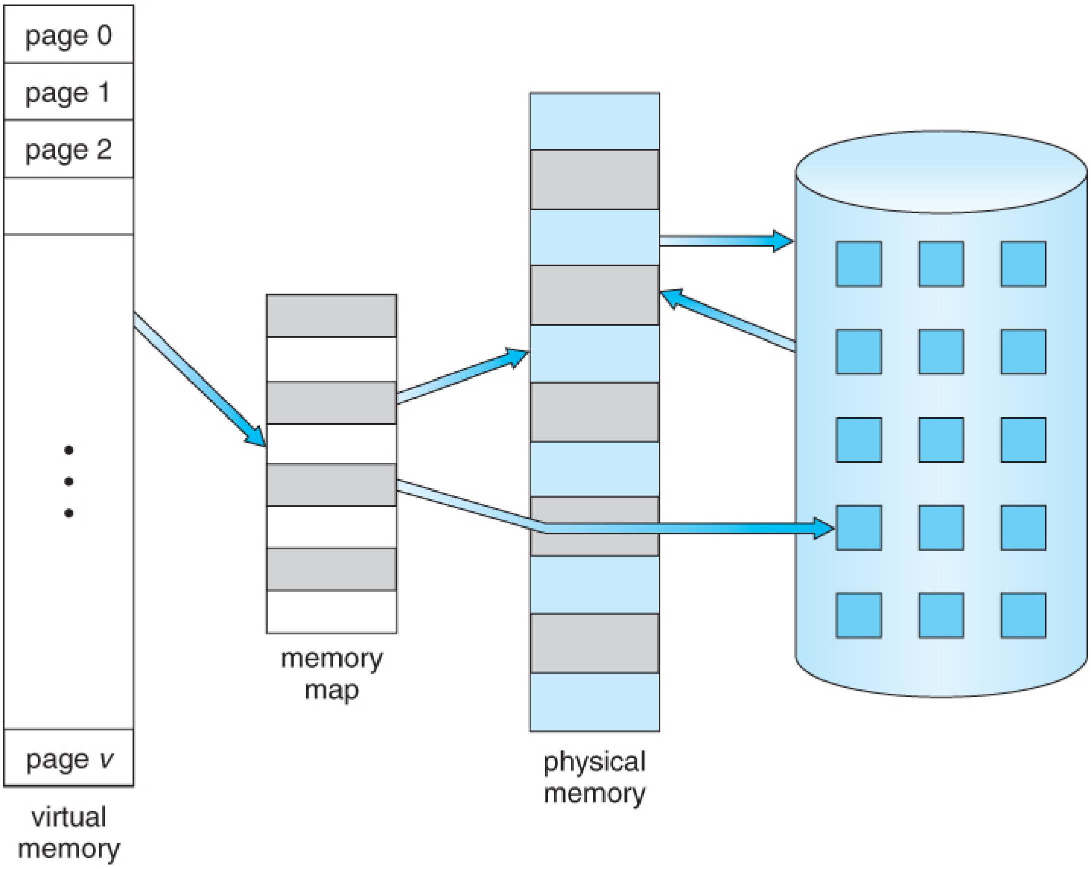


Figure 9.1 - Diagram showing virtual memory that is larger than physical memory

- Figure 9.2 shows ***virtual address space***, which is the programmers logical view of process memory storage. The actual physical layout is controlled by the process's page table.
- Note that the address space shown in Figure 9.2 is ***sparse*** - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.

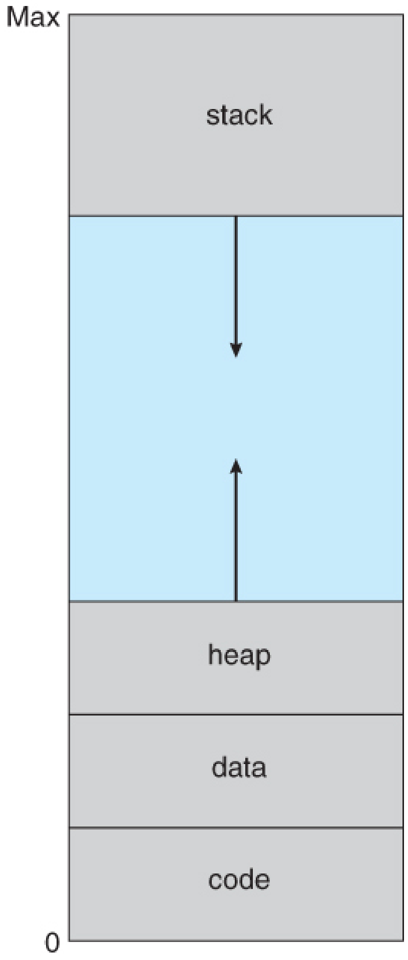


Figure 9.2 - Virtual address space

- Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits:
  - System libraries can be shared by mapping them into the virtual address space of more than one process.
  - Processes can also share virtual memory by mapping the same block of memory to more than one process.
  - Process pages can be shared during a fork( ) system call, eliminating the need to copy all of the pages of the original (parent) process.

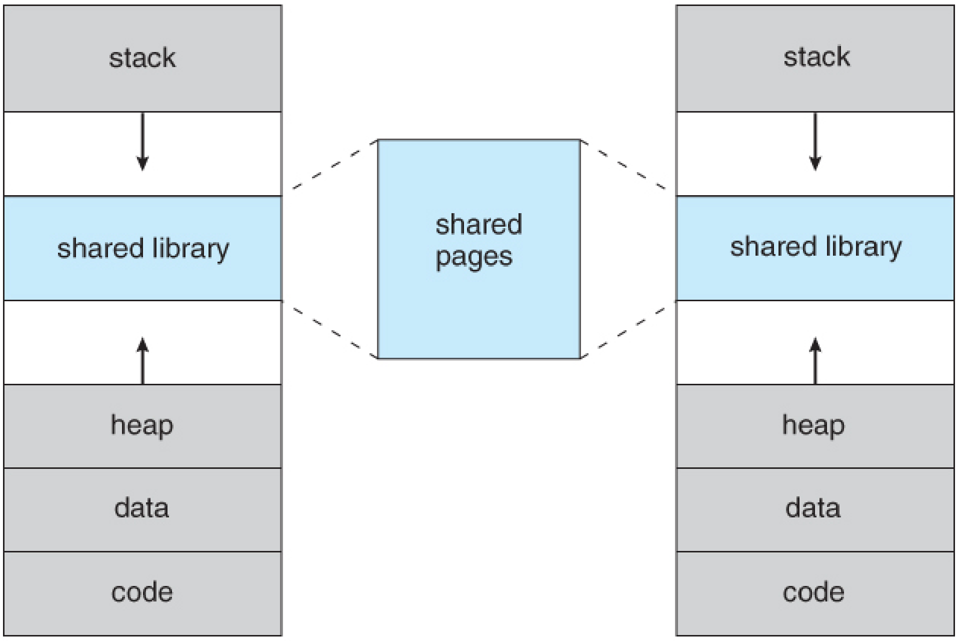


Figure 9.3 - Shared library using virtual memory

9.2 Demand Paging

- The basic idea behind ***demand paging*** is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them. (on demand.) This is termed a ***lazy swapper***, although a ***pager*** is a more accurate term.

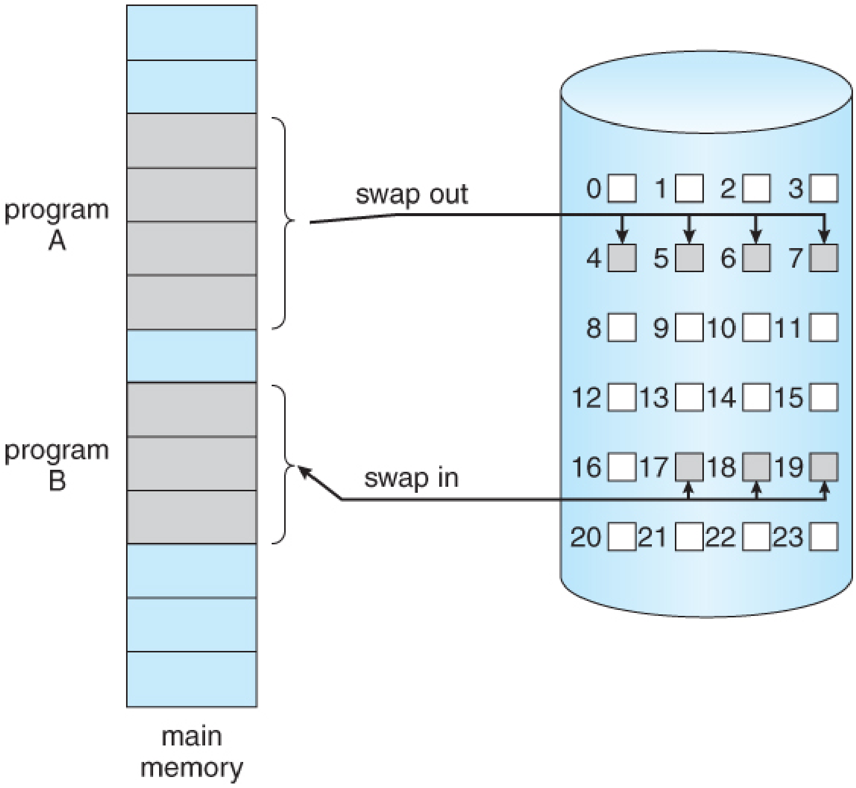


Figure 9.4 - Transfer of a paged memory to contiguous disk space

9.2.1 Basic Concepts

- The basic idea behind paging is that when a process is swapped in, the pager only loads into memory those pages that it expects the process to need (right away.)
- Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit. (The rest of the page table entry may either be blank or contain information about where to find the swapped-out page on the hard drive.)
- If the process only ever accesses pages that are loaded in memory (***memory resident*** pages), then the process runs exactly as if all the pages were loaded in to memory.

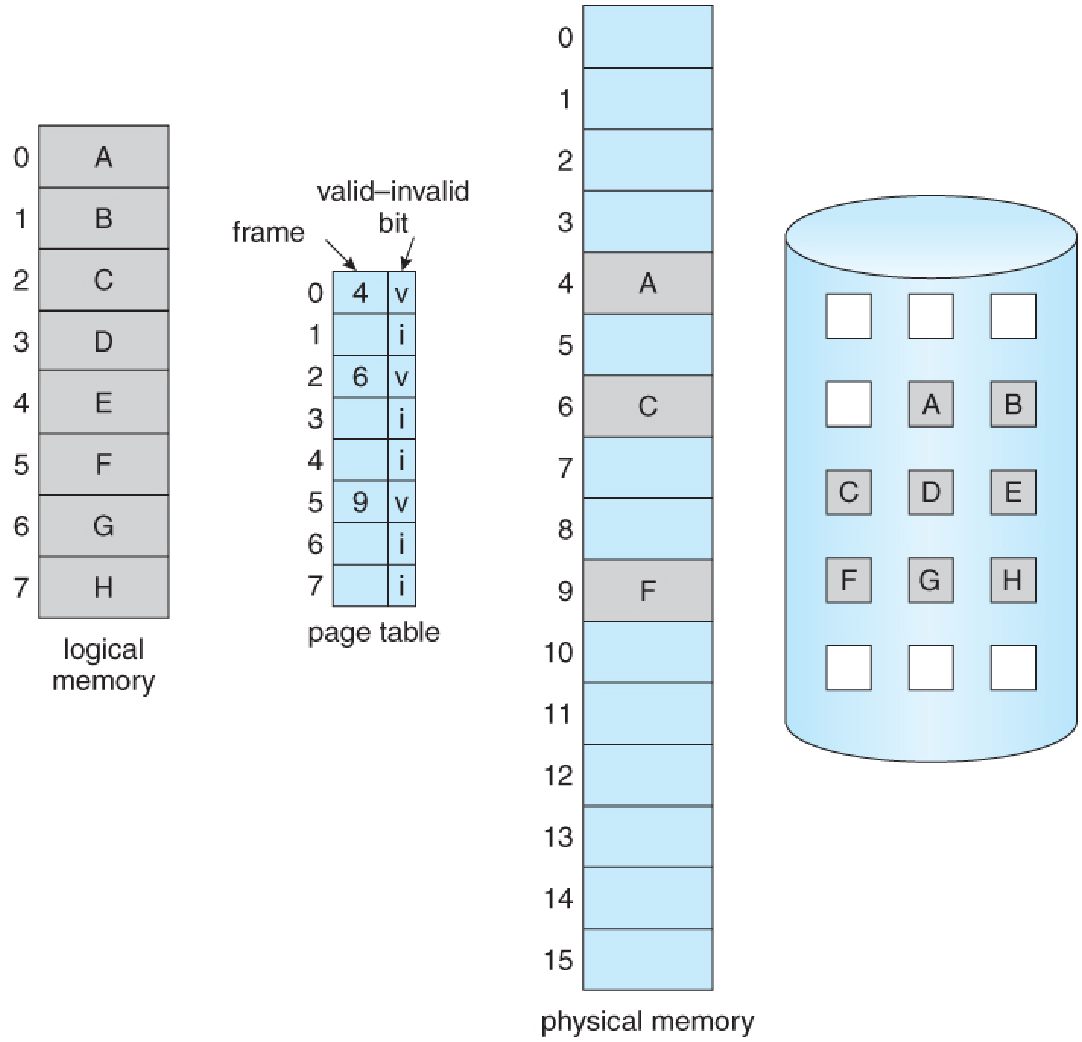


Figure 9.5 - Page table when some pages are not in main memory.

- On the other hand, if a page is needed that was not originally loaded up, then a ***page fault trap*** is generated, which must be handled in a series of steps:
    1. The memory address requested is first checked, to make sure it was a valid memory request.
    2. If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.
    3. A free frame is located, possibly from a free-frame list.
    4. A disk operation is scheduled to bring in the necessary page from disk. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.)
    5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
    6. The instruction that caused the page fault must now be restarted from the beginning, (as soon as this process gets another turn on the CPU.)

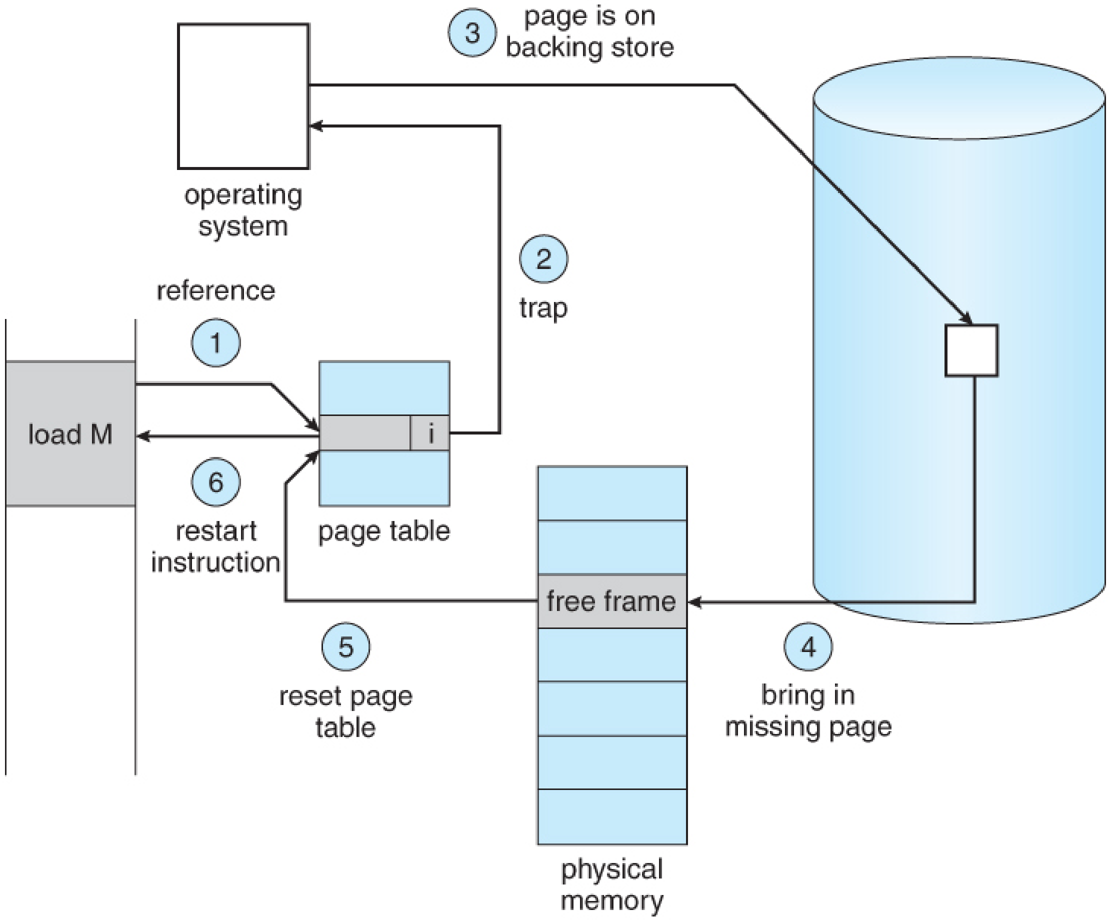


Figure 9.6 - Steps in handling a page fault

- In an extreme case, NO pages are swapped in for a process until they are requested by page faults. This is known as ***pure demand paging.***
- In theory each instruction could generate multiple page faults. In practice this is very rare, due to ***locality of reference***, covered in section 9.6.1.
- The hardware necessary to support virtual memory is the same as for paging and swapping: A page table and secondary memory. (***Swap space,*** whose allocation is discussed in chapter 12.)
- A crucial part of the process is that the instruction must be restarted from scratch once the desired page has been made available in memory. For most simple instructions this is not a major difficulty. However there are some architectures that allow a single instruction to modify a fairly large block of data, (which may span a page boundary), and if some of the data gets modified before the page fault occurs, this could cause problems. One solution is to access both ends of the block before executing the instruction, guaranteeing that the necessary pages get paged in before the instruction begins.

9.2.2 Performance of Demand Paging

- Obviously there is some slowdown and performance hit whenever a page fault occurs and the system has to go get it from memory, but just how big a hit is it exactly?
- There are many steps that occur when servicing a page fault (see book for full details), and some of the steps are optional or variable. But just for the sake of discussion, suppose that a normal memory access requires 200 nanoseconds, and that servicing a page fault takes 8 milliseconds. (8,000,000 nanoseconds, or 40,000 times a normal memory access.) With a ***page fault rate*** of p, (on a scale from 0 to 1), the effective access time is now:

$$(1 - p) * (200) + p * 8000000$$

$$= 200 + 7{,}999{,}800 * p$$

  which ***clearly*** depends heavily on p! Even if only one access in 1000 causes a page fault, the effective access time drops from 200 nanoseconds to 8.2 microseconds, a slowdown of a factor of 40 times. In order to keep the slowdown less than 10%, the page fault rate must be less than 0.0000025, or one in 399,990 accesses.

- A subtlety is that swap space is faster to access than the regular file system, because it does not have to go through the whole directory structure. For this reason some systems will transfer an entire process from the file system to swap space before starting up the process, so that future paging all occurs from the (relatively) faster swap space.
- Some systems use demand paging directly from the file system for binary code (which never changes and hence does not have to be stored on a page operation), and to reserve the swap space for data segments that must be stored. This approach is used by both Solaris and BSD Unix.

9.3 Copy-on-Write

- The idea behind a copy-on-write fork is that the pages for a parent process do not have to be actually copied for the child until one or the other of the processes changes the page. They can be simply shared between the two processes in the meantime, with a bit set that the page needs to be copied if it ever gets written to. This is a reasonable approach, since the child process usually issues an exec( ) system call immediately after the fork.

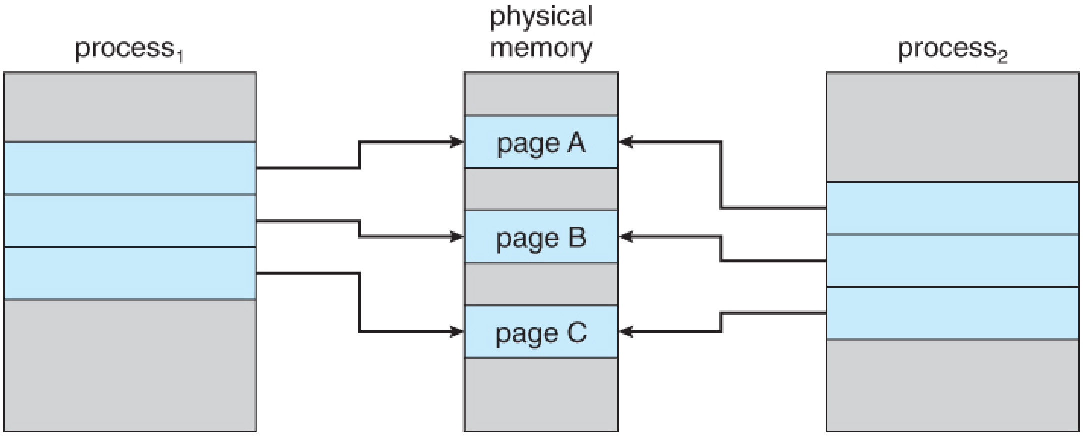


Figure 9.7 - Before process 1 modifies page C.

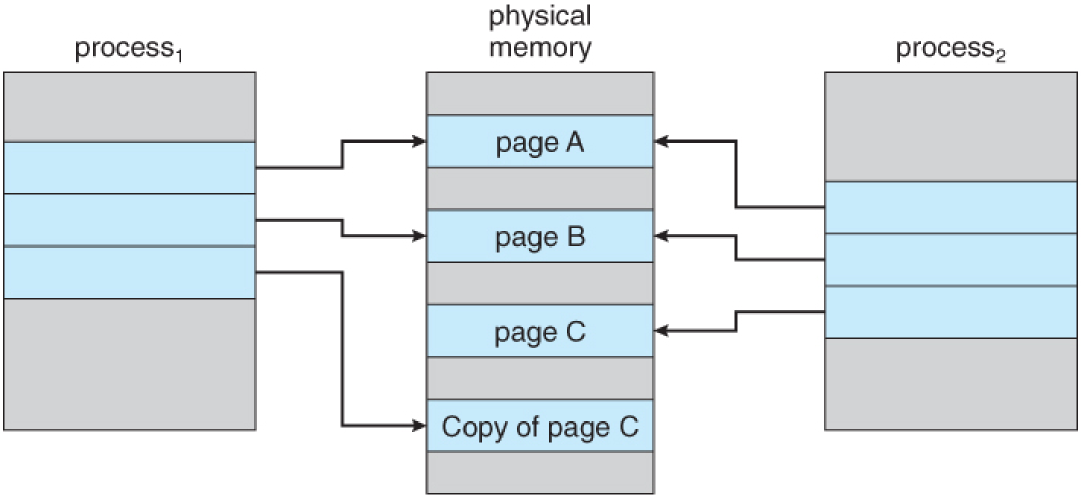


Figure 9.8 - After process 1 modifies page C.

- Obviously only pages that can be modified even need to be labeled as copy-on-write. Code segments can simply be shared.
- Pages used to satisfy copy-on-write duplications are typically allocated using ***zero-fill-on-demand***, meaning that their previous contents are zeroed out before the copy proceeds.
- Some systems provide an alternative to the fork( ) system call called a ***virtual memory fork, vfork( )***. In this case the parent is suspended, and the child uses the parent's memory pages. This is very fast for process creation, but requires that the child not modify any of the shared memory pages before performing the exec( ) system call. (In essence this addresses the question of which process executes first after a call to fork, the parent or the child. With vfork, the parent is suspended, allowing the child to execute first until it calls exec( ), sharing pages with the parent in the meantime.

9.4 Page Replacement

- In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load the pages that are actually needed by each process at any given time, there is room to load many more processes than if we had to load in the entire process.
- However memory is also needed for other purposes (such as I/O buffering), and what happens if some process suddenly decides it needs more pages and there aren't any free frames available? There are several possible solutions to consider:
  1. Adjust the memory used by I/O buffering, etc., to free up some frames for user processes. The decision of how to allocate memory for I/O versus user processes is a complex one, yielding different policies on different systems. (Some allocate a fixed amount for I/O, and others let the I/O system contend for memory along with everything else.)
  2. Put the process requesting more pages into a wait queue until some free frames become available.
  3. Swap some process out of memory completely, freeing up its page frames.
  4. Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. This is known as ***page replacement***, and is the most common solution. There are many different algorithms for page replacement, which is the subject of the remainder of this section.

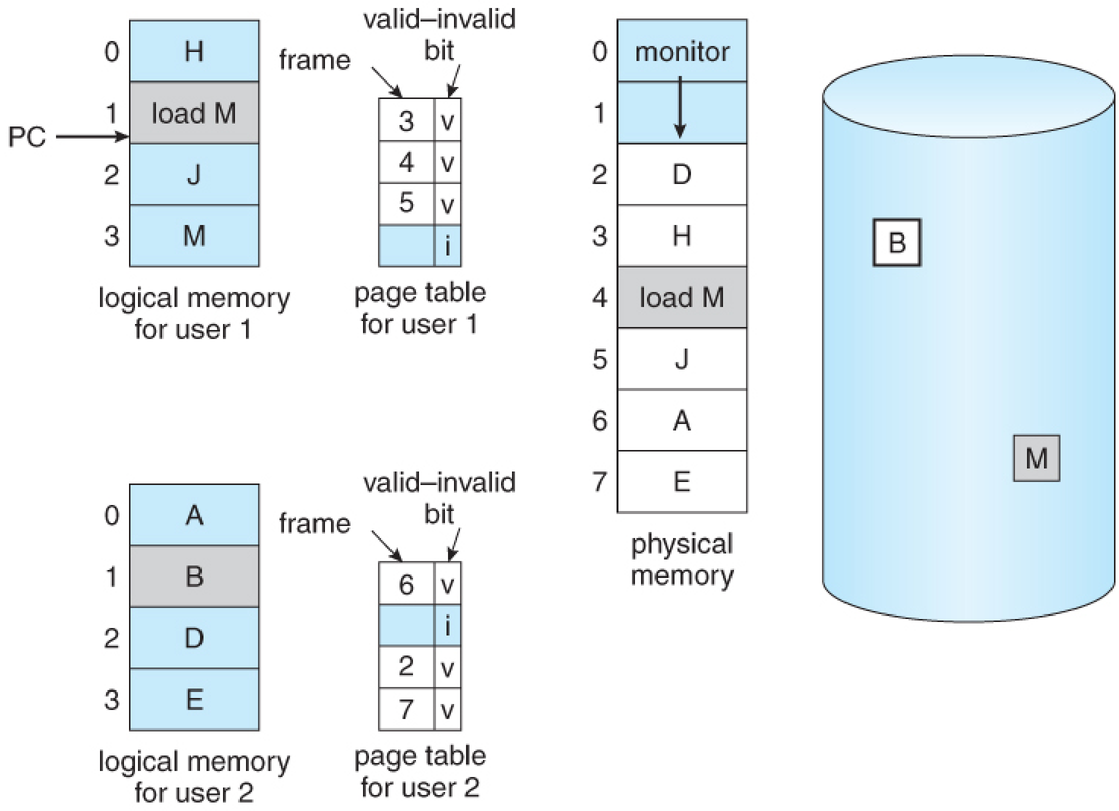


Figure 9.9 - Ned for page replacement.

9.4.1 Basic Page Replacement

- The previously discussed page-fault processing assumed that there would be free frames available on the free-frame list. Now the page-fault handling must be modified to free up a frame if necessary, as follows:
    1. Find the location of the desired page on the disk, either in swap space or in the file system.
    2. Find a free frame:
        a. If there is a free frame, use it.
        b. If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the ***victim frame***.
        c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
    3. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.
    4. Restart the process that was waiting for this page.

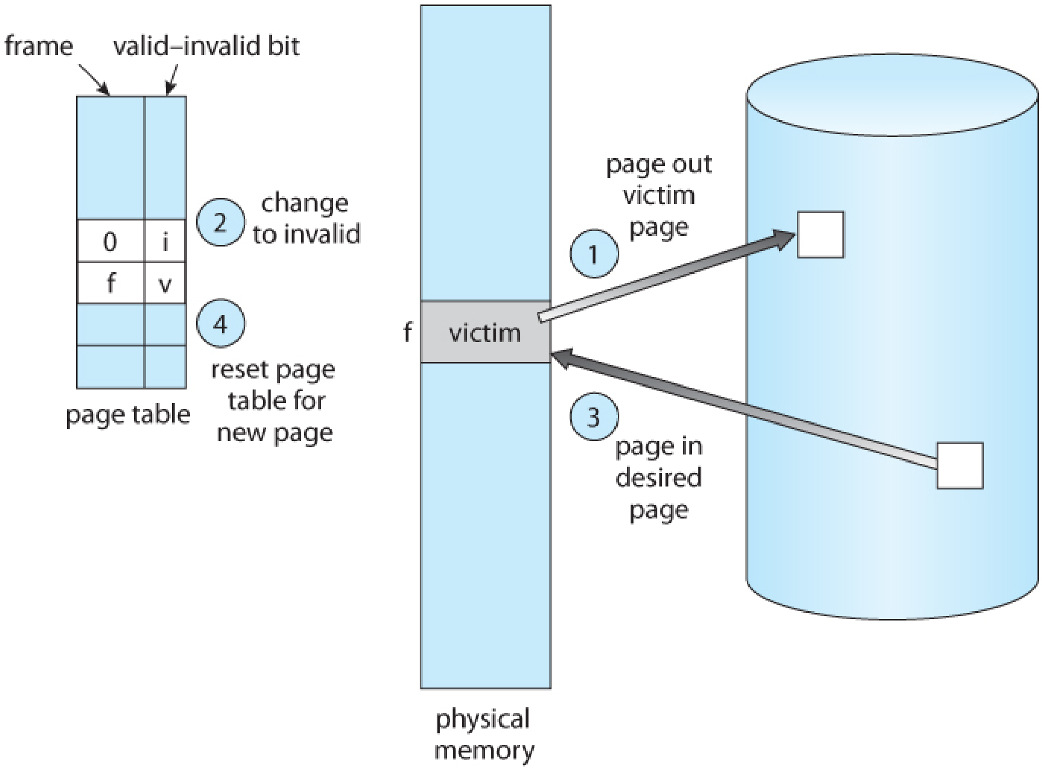


Figure 9.10 - Page replacement.

- Note that step 3c adds an extra disk write to the page-fault handling, effectively doubling the time required to process a page fault. This can be alleviated somewhat by assigning a ***modify bit,*** or ***dirty bit*** to each page, indicating whether or not it has been changed since it was last loaded in from disk. If the dirty bit has not been set, then the page is unchanged, and does not need to be written out to disk. Otherwise the page write is required. It should come as no surprise that many page replacement strategies specifically look for pages that do not have their dirty bit set, and preferentially select clean pages as victim pages. It should also be obvious that unmodifiable code pages never get their dirty bits set.
- There are two major requirements to implement a successful demand paging system. We must develop a ***frame-allocation algorithm*** and a ***page-replacement algorithm.*** The former centers around how many frames are allocated to each process (and to other needs), and the latter deals with how to select a page for replacement when there are no free frames available.
- The overall goal in selecting and tuning these algorithms is to generate the fewest number of overall page faults. Because disk access is so slow relative to memory access, even slight improvements to these algorithms can yield large improvements in overall system performance.
- Algorithms are evaluated using a given string of memory accesses known as a ***reference string,*** which can be generated in one of (at least) three common ways:
    1. Randomly generated, either evenly distributed or with some distribution curve based on observed system behavior. This is the fastest and easiest approach, but may not reflect real performance well, as it ignores locality of reference.
    2. Specifically designed sequences. These are useful for illustrating the properties of comparative algorithms in published papers and textbooks, (and also for homework and exam problems. :-))
    3. Recorded memory references from a live system. This may be the best approach, but the amount of data collected can be enormous, on the order of a million addresses per second. The volume of collected data can be reduced by making two important observations:
        1. Only the page number that was accessed is relevant. The offset within that page does not affect paging operations.
        2. Successive accesses within the same page can be treated as a single page request, because all requests after the first are guaranteed to be page hits. (Since there are no intervening requests for other pages that could remove this page from the page table.)
        - So for example, if pages were of size 100 bytes, then the sequence of address requests (0100, 0432, 0101, 0612, 0634, 0688, 0132, 0038, 0420) would reduce to page requests (1, 4, 1, 6, 1, 0, 4)
- As the number of available frames increases, the number of page faults should decrease, as shown in Figure 9.11:

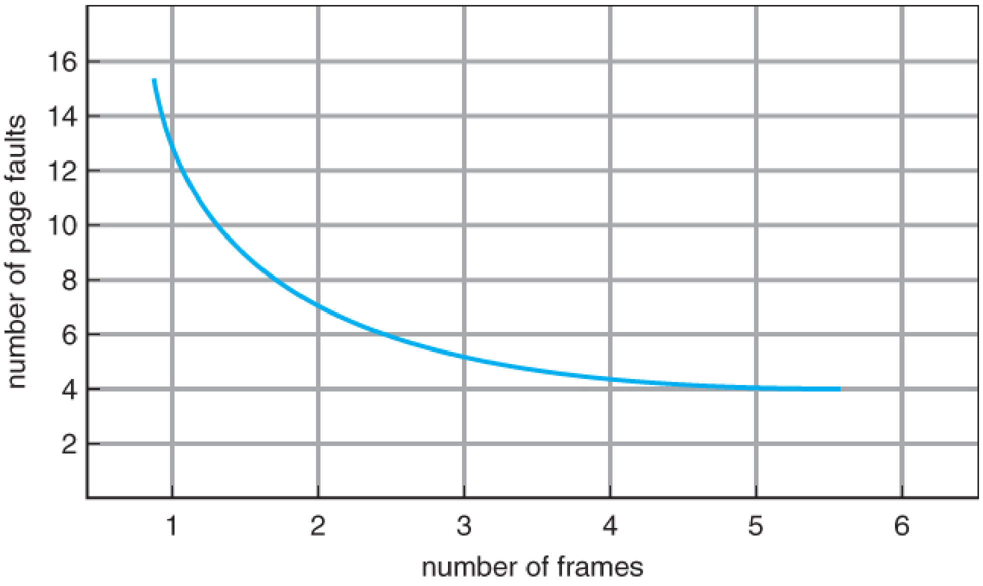


Figure 9.11 - Graph of page faults versus number of frames.

9.4.2 FIFO Page Replacement

- A simple and obvious page replacement strategy is ***FIFO***, i.e. first-in-first-out.
- As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim. In the following example, 20 page requests result in 15 page faults:

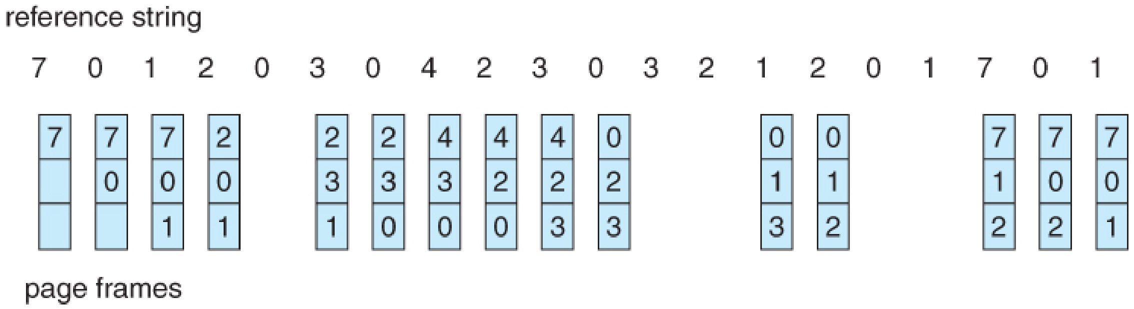


Figure 9.12 - FIFO page-replacement algorithm.

- Although FIFO is simple and easy, it is not always optimal, or even efficient.
- An interesting effect that can occur with FIFO is ***Belady's anomaly***, in which increasing the number of frames available can actually ***increase*** the number of page faults that occur! Consider, for example, the following chart based on the page sequence

(1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5) and a varying number of available frames. Obviously the maximum number of faults is 12 (every request generates a fault), and the minimum number is 5 (each page loaded only once), but in between there are some interesting results:

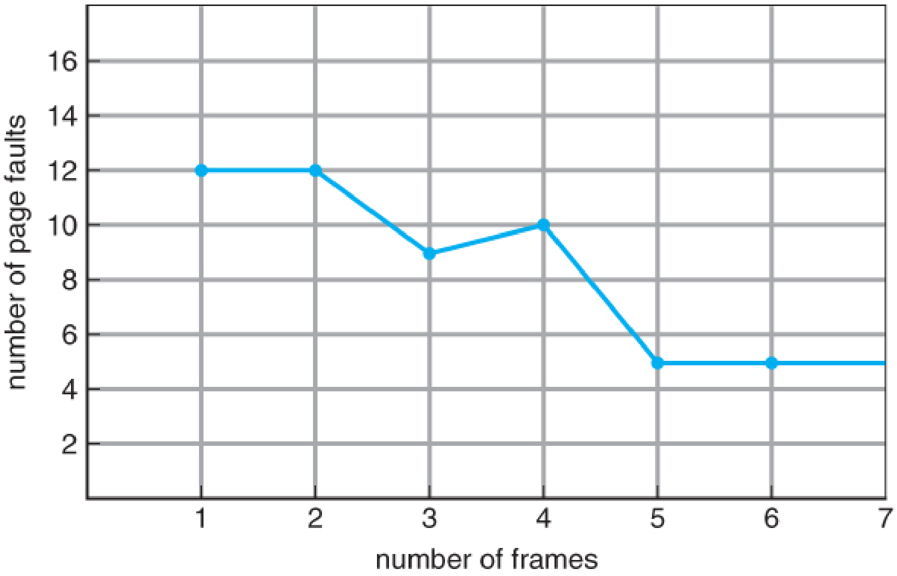


Figure 9.13 - Page-fault curve for FIFO replacement on a reference string.

9.4.3 Optimal Page Replacement

- The discovery of Belady's anomaly lead to the search for an ***optimal page-replacement algorithm***, which is simply that which yields the lowest of all possible page-faults, and which does not suffer from Belady's anomaly.
- Such an algorithm does exist, and is called ***OPT or MIN.*** This algorithm is simply "Replace the page that will not be used for the longest time in the future."
- For example, Figure 9.14 shows that by applying OPT to the same reference string used for the FIFO example, the minimum number of possible page faults is 9. Since 6 of the page-faults are unavoidable (the first reference to each new page), FIFO can be shown to require 3 times as many (extra) page faults as the optimal algorithm. (Note: The book claims that only the first three page faults are required by all algorithms, indicating that FIFO is only twice as bad as OPT.)
- Unfortunately OPT cannot be implemented in practice, because it requires foretelling the future, but it makes a nice benchmark for the comparison and evaluation of real proposed new algorithms.
- In practice most page-replacement algorithms try to approximate OPT by predicting (estimating) in one fashion or another what page will not be used for the longest period of time. The basis of FIFO is the prediction that the page that was brought in the longest time ago is the one that will not be needed again for the longest future time, but as we shall see, there are many other prediction methods, all striving to match the performance of OPT.

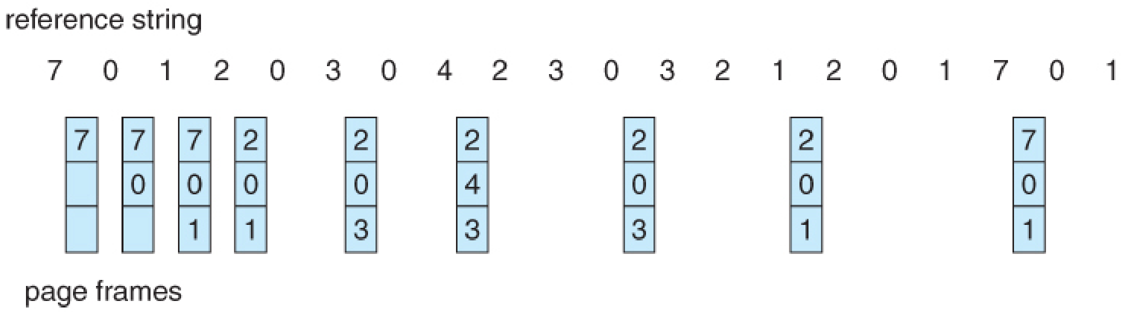


Figure 9.14 - Optimal page-replacement algorithm

9.4.4 LRU Page Replacement

- The prediction behind ***LRU,*** the ***Least Recently Used,*** algorithm is that the page that has not been used in the longest time is the one that will not be used again in the near future. (Note the distinction between FIFO and LRU: The former looks at the oldest ***load*** time, and the latter looks at the oldest ***use*** time.)
- Some view LRU as analogous to OPT, except looking backwards in time instead of forwards. (OPT has the interesting property that for any reference string S and its reverse R, OPT will generate the same number of page faults for S and for R. It turns out that LRU has this same property.)
- Figure 9.15 illustrates LRU for our sample string, yielding 12 page faults, (as compared to 15 for FIFO and 9 for OPT.)

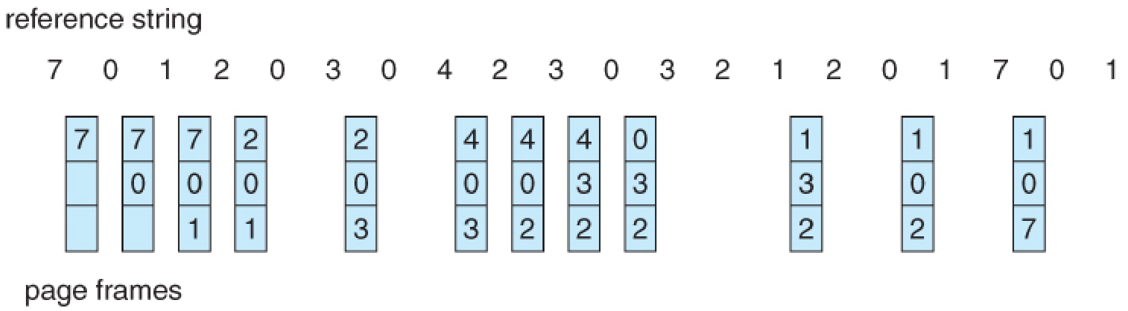


Figure 9.15 - LRU page-replacement algorithm.

- LRU is considered a good replacement policy, and is often used. The problem is how exactly to implement it. There are two simple approaches commonly used:
  1. **Counters.** Every memory access increments a counter, and the current value of this counter is stored in the page table entry for that page. Then finding the LRU page involves simple searching the table for the page with the smallest counter value. Note that overflowing of the counter must be considered.
  2. **Stack.** Another approach is to use a stack, and whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.
- Note that both implementations of LRU require hardware support, either for incrementing the counter or for managing the stack, as these operations must be performed for ***every*** memory access.
- Neither LRU or OPT exhibit Belady's anomaly. Both belong to a class of page-replacement algorithms called ***stack algorithms,*** which can never exhibit Belady's anomaly. A stack algorithm is one in which the pages kept in memory for a frame set of size N will always be a subset of the pages kept for a frame size of N + 1. In the case of LRU, (and particularly the stack implementation thereof), the top N pages of the stack will be the same for all frame set sizes of N or anything larger.

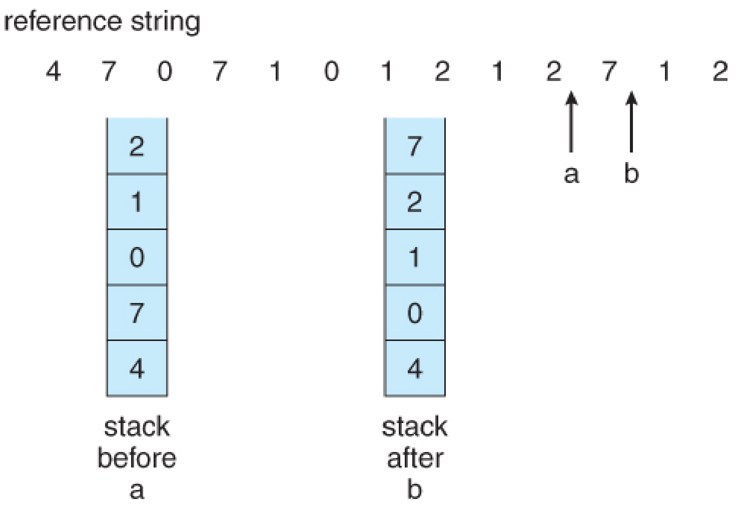


Figure 9.16 - Use of a stack to record the most recent page references.

9.4.5 LRU-Approximation Page Replacement

- Unfortunately full implementation of LRU requires hardware support, and few systems provide the full hardware support necessary.
- However many systems offer some degree of HW support, enough to approximate LRU fairly well. (In the absence of ANY hardware support, FIFO might be the best available choice.)
- In particular, many systems provide a ***reference bit*** for every entry in a page table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time. One bit of precision is enough to distinguish pages that have been accessed since the last clear from those that have not, but does not provide any finer grain of detail.

9.4.5.1 Additional-Reference-Bits Algorithm

- Finer grain is possible by storing the most recent 8 reference bits for each page in an 8-bit byte in the page table entry, which is interpreted as an unsigned int.
  - At periodic intervals (clock interrupts), the OS takes over, and right-shifts each of the reference bytes by one bit.
  - The high-order (leftmost) bit is then filled in with the current value of the reference bit, and the reference bits are cleared.
  - At any given time, the page with the smallest value for the reference byte is the LRU page.
- Obviously the specific number of bits used and the frequency with which the reference byte is updated are adjustable, and are tuned to give the fastest performance on a given hardware platform.

9.4.5.2 Second-Chance Algorithm

- The ***second chance algorithm*** is essentially a FIFO, except the reference bit is used to give pages a second chance at staying in the page table.
    - When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner.
    - If a page is found with its reference bit not set, then that page is selected as the next victim.
    - If, however, the next page in the FIFO **does** have its reference bit set, then it is given a second chance:
        - The reference bit is cleared, and the FIFO search continues.
        - If some other page is found that did not have its reference bit set, then that page will be selected as the victim, and this page (the one being given the second chance) will be allowed to stay in the page table.
        - If , however, there are no other pages that do not have their reference bit set, then this page will be selected as the victim when the FIFO search circles back around to this page on the second pass.
- If all reference bits in the table are set, then second chance degrades to FIFO, but also requires a complete search of the table for every page-replacement.
- As long as there are some pages whose reference bits are not set, then any page referenced frequently enough gets to stay in the page table indefinitely.
- This algorithm is also known as the ***clock*** algorithm, from the hands of the clock moving around the circular queue.

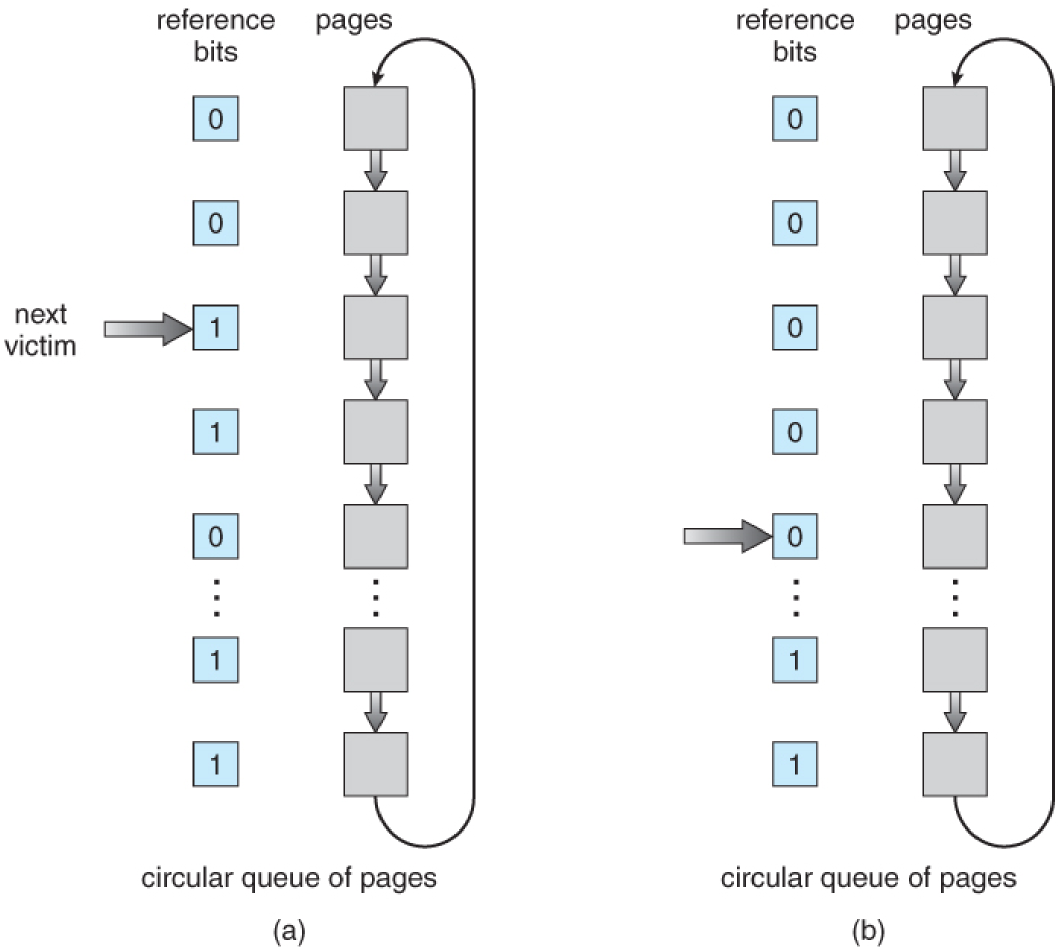


Figure 9.17 - Second-chance (clock) page-replacement algorithm.

9.4.5.3 Enhanced Second-Chance Algorithm

- The **enhanced second chance algorithm** looks at the reference bit and the modify bit (dirty bit) as an ordered page, and classifies pages into one of four classes:
    1. (0, 0) - Neither recently used nor modified.
    2. (0, 1) - Not recently used, but modified.
    3. (1, 0) - Recently used, but clean.
    4. (1, 1) - Recently used and modified.
- This algorithm searches the page table in a circular fashion (in as many as four passes), looking for the first page it can find in the lowest numbered category. I.e. it first makes a pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a (0, 1), etc.
- The main difference between this algorithm and the previous one is the preference for replacing clean pages if possible.

9.4.6 Counting-Based Page Replacement

- There are several algorithms based on counting the number of references that have been made to a given page, such as:
    - ***Least Frequently Used, LFU:*** Replace the page with the lowest reference count. A problem can occur if a page is used frequently initially and then not used any more, as the reference count remains high. A solution to this problem is to right-shift the counters periodically, yielding a time-decaying average reference count.

    - ***Most Frequently Used, MFU:*** Replace the page with the highest reference count. The logic behind this idea is that pages that have already been referenced a lot have been in the system a long time, and we are probably done with them, whereas pages referenced only a few times have only recently been loaded, and we still need them.
- In general counting-based algorithms are not commonly used, as their implementation is expensive and they do not approximate OPT well.

9.4.7 Page-Buffering Algorithms

There are a number of page-buffering algorithms that can be used in conjunction with the afore-mentioned algorithms, to improve overall performance and sometimes make up for inherent weaknesses in the hardware and/or the underlying page-replacement algorithms:

- Maintain a certain minimum number of free frames at all times. When a page-fault occurs, go ahead and allocate one of the free frames from the free list first, to get the requesting process up and running again as quickly as possible, and then select a victim page to write to disk and free up a frame as a second step.
- Keep a list of modified pages, and when the I/O system is otherwise idle, have it write these pages out to disk, and then clear the modify bits, thereby increasing the chance of finding a "clean" page for the next potential victim.
- Keep a pool of free frames, but remember what page was in it before it was made free. Since the data in the page is not actually cleared out when the page is freed, it can be made an active page again without having to load in any new data from disk. This is useful when an algorithm mistakenly replaces a page that in fact is needed again soon.

9.4.8 Applications and Page Replacement

- Some applications (most notably database programs) understand their data accessing and caching needs better than the general-purpose OS, and should therefore be given reign to do their own memory management.
- Sometimes such programs are given a ***raw disk partition*** to work with, containing raw data blocks and no file system structure. It is then up to the application to use this disk partition as extended memory or for whatever other reasons it sees fit.

9.5 Allocation of Frames

We said earlier that there were two important tasks in virtual memory management: a page-replacement strategy and a frame-allocation strategy. This section covers the second part of that pair.

9.5.1 Minimum Number of Frames

- The absolute minimum number of frames that a process must be allocated is dependent on system architecture, and corresponds to the worst-case scenario of the number of pages that could be touched by a single (machine) instruction.
- If an instruction (and its operands) spans a page boundary, then multiple pages could be needed just for the instruction fetch.
- Memory references in an instruction touch more pages, and if those memory locations can span page boundaries, then multiple pages could be needed for operand access also.
- The worst case involves indirect addressing, particularly where multiple levels of indirect addressing are allowed. Left unchecked, a pointer to a pointer to a pointer to a pointer to a . . . could theoretically touch every page in the virtual address space in a single machine instruction, requiring every virtual page be loaded into physical memory simultaneously. For this reason architectures place a limit (say 16) on the number of levels of indirection allowed in an instruction, which is enforced with a counter initialized to the limit and decremented with every level of indirection in an instruction - If the counter reaches zero, then an "excessive indirection" trap occurs. This example would still require a minimum frame allocation of 17 per process.

9.5.2 Allocation Algorithms

- **Equal Allocation -** If there are m frames available and n processes to share them, each process gets m / n frames, and the leftovers are kept in a free-frame buffer pool.
- **Proportional Allocation -** Allocate the frames proportionally to the size of the process, relative to the total size of all processes. So if the size of process i is S_i, and S is the sum of all S_i, then the allocation for process P_i is a_i = m * S_i / S.
- Variations on proportional allocation could consider priority of process rather than just their size.
- Obviously all allocations fluctuate over time as the number of available free frames, m, fluctuates, and all are also subject to the constraints of minimum allocation. (If the minimum allocations cannot be met, then processes must either be swapped out or not allowed to start until more free frames become available.)

9.5.3 Global versus Local Allocation

- One big question is whether frame allocation (page replacement) occurs on a local or global level.
- With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process.
- With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not.
- Local page replacement allows processes to better control their own page fault rates, and leads to more consistent performance of a given process over different system load levels.
- Global page replacement is overall more efficient, and is the more commonly used approach.

9.5.4 Non-Uniform Memory Access

- The above arguments all assume that all memory is equivalent, or at least has equivalent access times.

- This may not be the case in multiple-processor systems, especially where each CPU is physically located on a separate circuit board which also holds some portion of the overall system memory.
- In these latter systems, CPUs can access memory that is physically located on the same board much faster than the memory on the other boards.
- The basic solution is akin to processor affinity - At the same time that we try to schedule processes on the same CPU to minimize cache misses, we also try to allocate memory for those processes on the same boards, to minimize access times.
- The presence of threads complicates the picture, especially when the threads get loaded onto different processors.
- Solaris uses an ***lgroup*** as a solution, in a hierarchical fashion based on relative latency. For example, all processors and RAM on a single board would probably be in the same lgroup. Memory assignments are made within the same lgroup if possible, or to the next nearest lgroup otherwise. (Where "nearest" is defined as having the lowest access time.)

9.6 Thrashing

- If a process cannot maintain its minimum required number of frames, then it must be swapped out, freeing up frames for other processes. This is an intermediate level of CPU scheduling.
- But what about a process that can keep its minimum, but cannot keep all of the frames that it is currently using on a regular basis? In this case it is forced to page out pages that it will need again in the very near future, leading to large numbers of page faults.
- A process that is spending more time paging than executing is said to be ***thrashing.***

9.6.1 Cause of Thrashing

- Early process scheduling schemes would control the level of multiprogramming allowed based on CPU utilization, adding in more processes when CPU utilization was low.
- The problem is that when memory filled up and processes started spending lots of time waiting for their pages to page in, then CPU utilization would lower, causing the schedule to add in even more processes and exacerbating the problem! Eventually the system would essentially grind to a halt.
- Local page replacement policies can prevent one thrashing process from taking pages away from other processes, but it still tends to clog up the I/O queue, thereby slowing down any other process that needs to do even a little bit of paging (or any other I/O for that matter.)

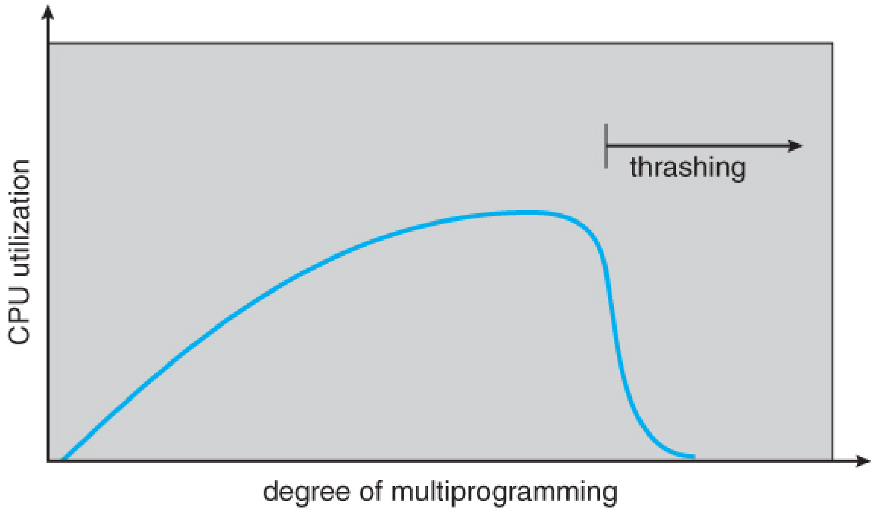


Figure 9.18 - Thrashing

- To prevent thrashing we must provide processes with as many frames as they really need "right now", but how do we know what that is?
- The ***locality model*** notes that processes typically access memory references in a given ***locality,*** making lots of references to the same general area of memory before moving periodically to a new locality, as shown in Figure 9.19 below. If we could just keep as many frames as are involved in the current locality, then page faulting would occur primarily on switches from one locality to another. (E.g. when one function exits and another is called.)

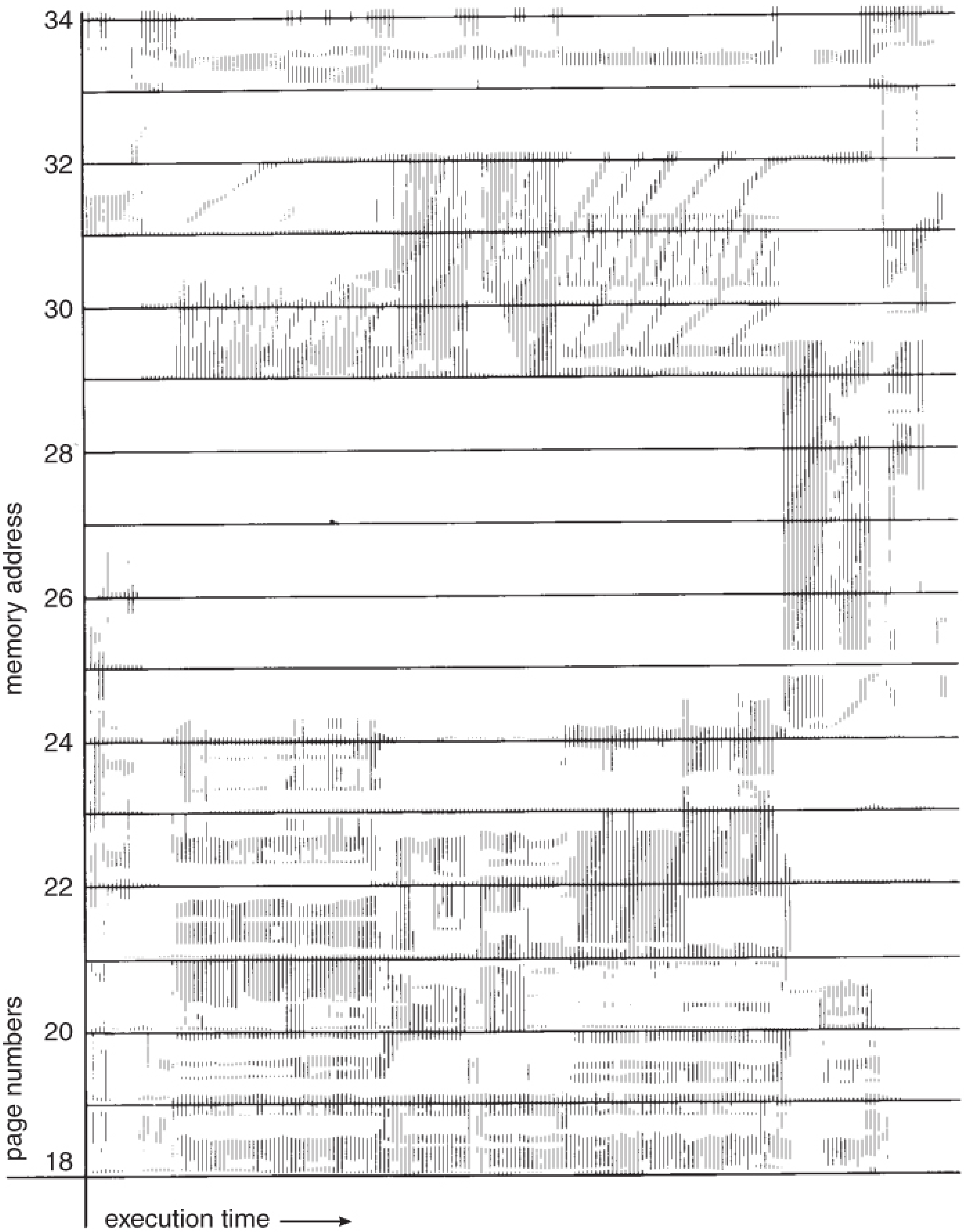


Figure 9.19 - Locality in a memory-reference pattern.

9.6.2 Working-Set Model

- The ***working set model*** is based on the concept of locality, and defines a ***working set window***, of length ***delta.*** Whatever pages are included in the most recent delta page references are said to be in the processes working set window, and comprise its current working set, as illustrated in Figure 9.20:

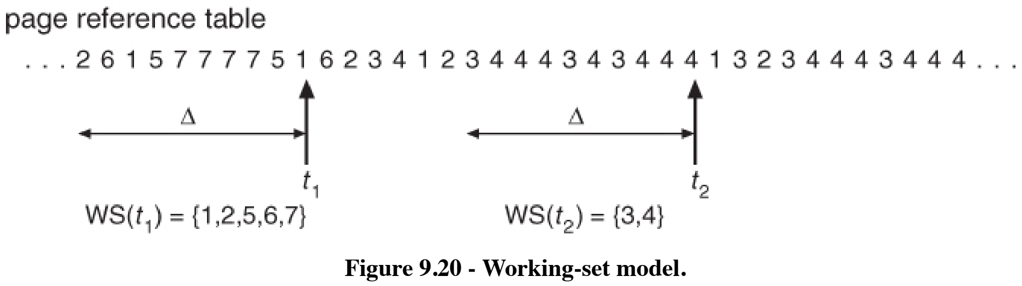


Figure 9.20 - Working-set model.

- The selection of delta is critical to the success of the working set model - If it is too small then it does not encompass all of the pages of the current locality, and if it is too large, then it encompasses pages that are no longer being frequently accessed.
- The total demand, D, is the sum of the sizes of the working sets for all processes. If D exceeds the total number of available frames, then at least one process is thrashing, because there are not enough frames available to satisfy its minimum working set. If D is significantly less than the currently available frames, then additional processes can be launched.

- The hard part of the working-set model is keeping track of what pages are in the current working set, since every reference adds one to the set and removes one older page. An approximation can be made using reference bits and a timer that goes off after a set interval of memory references:
    - For example, suppose that we set the timer to go off after every 5000 references (by any process), and we can store two additional historical reference bits in addition to the current reference bit.
    - Every time the timer goes off, the current reference bit is copied to one of the two historical bits, and then cleared.
    - If any of the three bits is set, then that page was referenced within the last 15,000 references, and is considered to be in that processes reference set.
    - Finer resolution can be achieved with more historical bits and a more frequent timer, at the expense of greater overhead.

9.6.3 Page-Fault Frequency

- A more direct approach is to recognize that what we really want to control is the page-fault rate, and to allocate frames based on this directly measurable value. If the page-fault rate exceeds a certain upper bound then that process needs more frames, and if it is below a given lower bound, then it can afford to give up some of its frames to other processes.
- (I suppose a page-replacement strategy could be devised that would select victim frames based on the process with the lowest current page-fault frequency.)

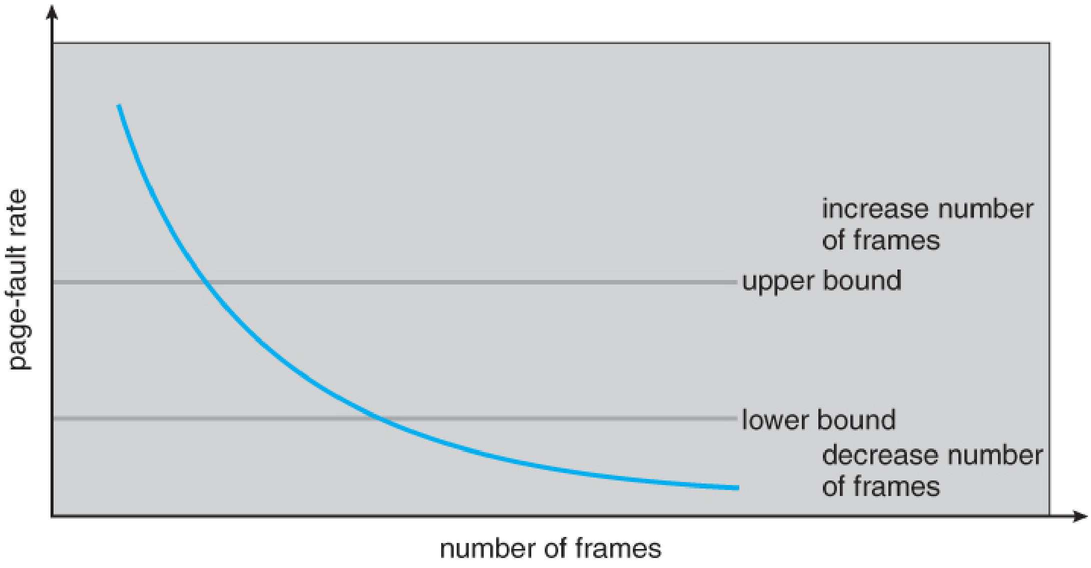
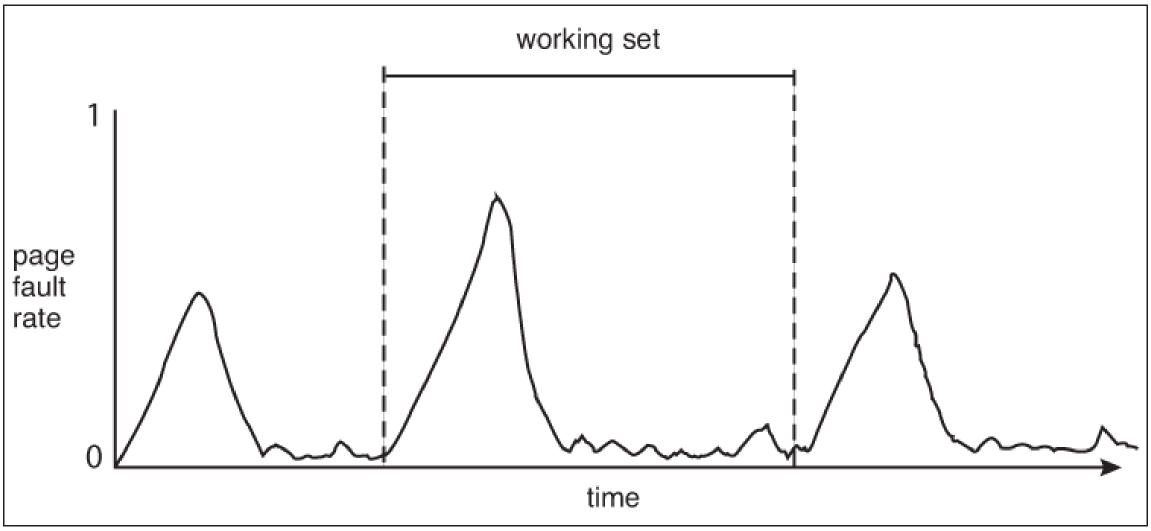


Figure 9.21 - Page-fault frequency.

- Note that there is a direct relationship between the page-fault rate and the working-set, as a process moves from one locality to another:



Unnumbered side bar in Ninth Edition

9.7 Memory-Mapped Files

- Rather than accessing data files directly via the file system with every file access, data files can be paged into memory the same as process files, resulting in much faster accesses (except of course when page-faults occur.) This is known as ***memory-mapping*** a file.

9.7.1 Basic Mechanism

- Basically a file is mapped to an address range within a process's virtual address space, and then paged in as needed using the ordinary demand paging system.

- Note that file writes are made to the memory page frames, and are not immediately written out to disk. (This is the purpose of the "flush( )" system call, which may also be needed for stdout in some cases. See the timekiller program for an example of this.)
- This is also why it is important to "close( )" a file when one is done writing to it - So that the data can be safely flushed out to disk and so that the memory frames can be freed up for other purposes.
- Some systems provide special system calls to memory map files and use direct disk access otherwise. Other systems map the file to process address space if the special system calls are used and map the file to kernel address space otherwise, but do memory mapping in either case.
- File sharing is made possible by mapping the same file to the address space of more than one process, as shown in Figure 9.23 below. Copy-on-write is supported, and mutual exclusion techniques (chapter 6) may be needed to avoid synchronization problems.

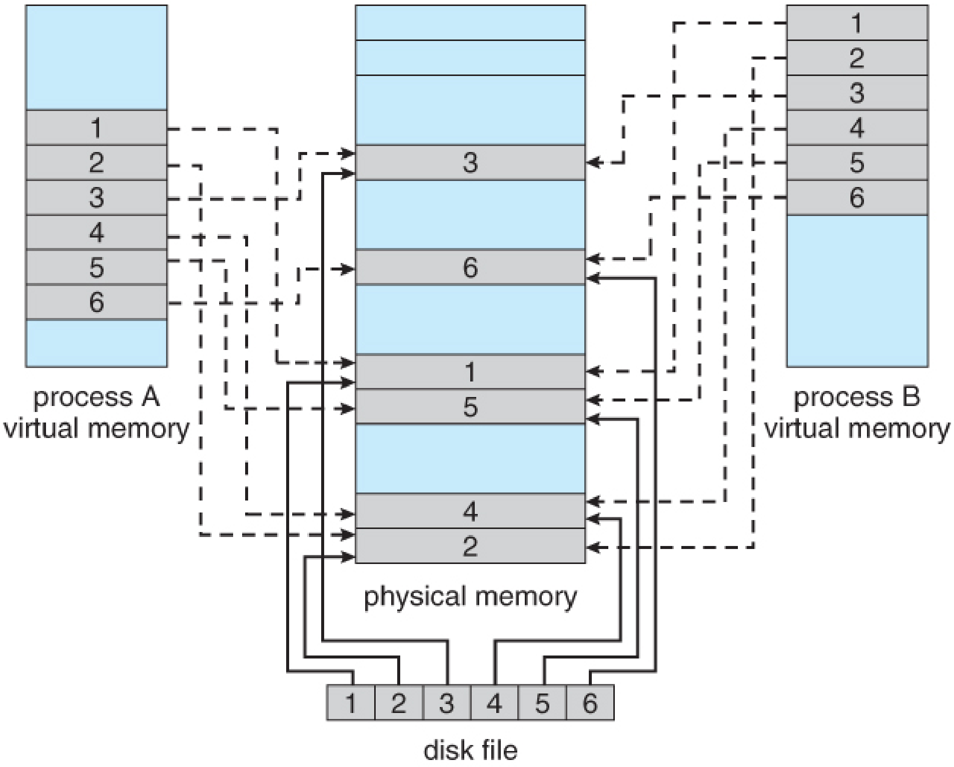


Figure 9.22 Memory-mapped files.

- Shared memory can be implemented via shared memory-mapped files (Windows), or it can be implemented through a separate process (Linux, UNIX.)

9.7.2 Shared Memory in the Win32 API

- Windows implements shared memory using shared memory-mapped files, involving three basic steps:
    1. Create a file, producing a HANDLE to the new file.
    2. Name the file as a shared object, producing a HANDLE to the shared object.
    3. Map the shared object to virtual memory address space, returning its base address as a void pointer (LPVOID).
- This is illustrated in Figures 9.24 to 9.26 (annotated.)

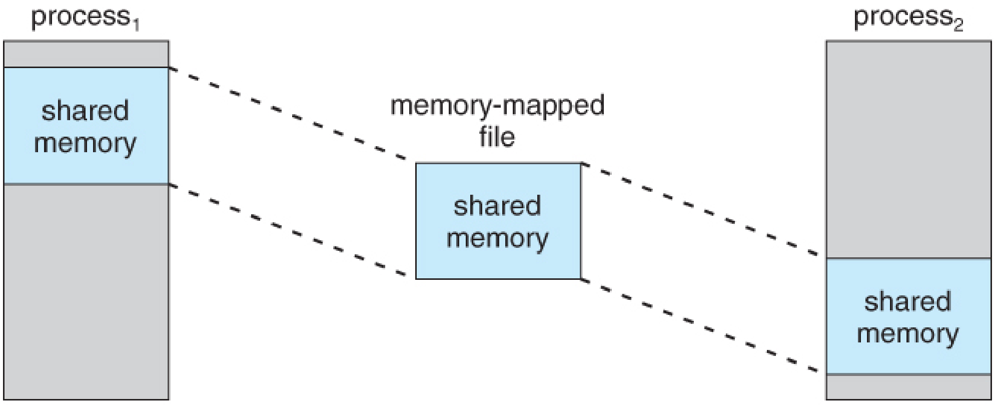
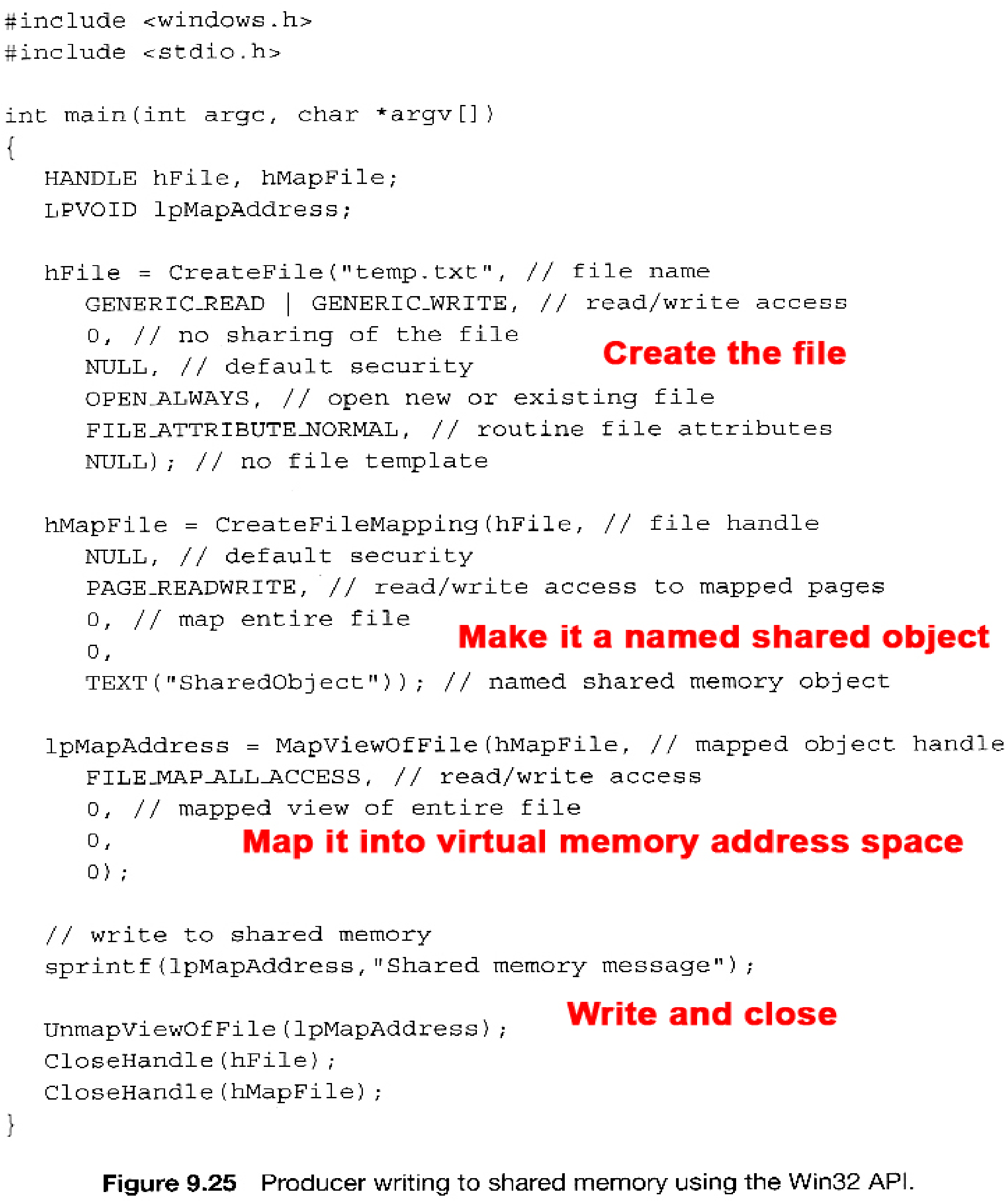


Figure 9.23 - Shared memory in Windows using memory-mapped I/O.

Producer creates file, makes it a shared object, maps it to memory, writes to it, and closes



```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   HANDLE hFile, hMapFile;
   LPVOID lpMapAddress;

   hFile = CreateFile("temp.txt", // file name
      GENERIC_READ | GENERIC_WRITE, // read/write access
      0, // no sharing of the file
      NULL, // default security
      OPEN_ALWAYS, // open new or existing file
      FILE_ATTRIBUTE_NORMAL, // routine file attributes
      NULL); // no file template

   hMapFile = CreateFileMapping(hFile, // file handle
      NULL, // default security
      PAGE_READWRITE, // read/write access to mapped pages
      0, // map entire file
      0,
      TEXT("SharedObject")); // named shared memory object

   lpMapAddress = MapViewOfFile(hMapFile, // mapped object handle
      FILE_MAP_ALL_ACCESS, // read/write access
      0, // mapped view of entire file
      0,
      0);

   // write to shared memory
   sprintf(lpMapAddress,"Shared memory message");

   UnmapViewOfFile(lpMapAddress);
   CloseHandle(hFile);
   CloseHandle(hMapFile);
}
```

Create the file

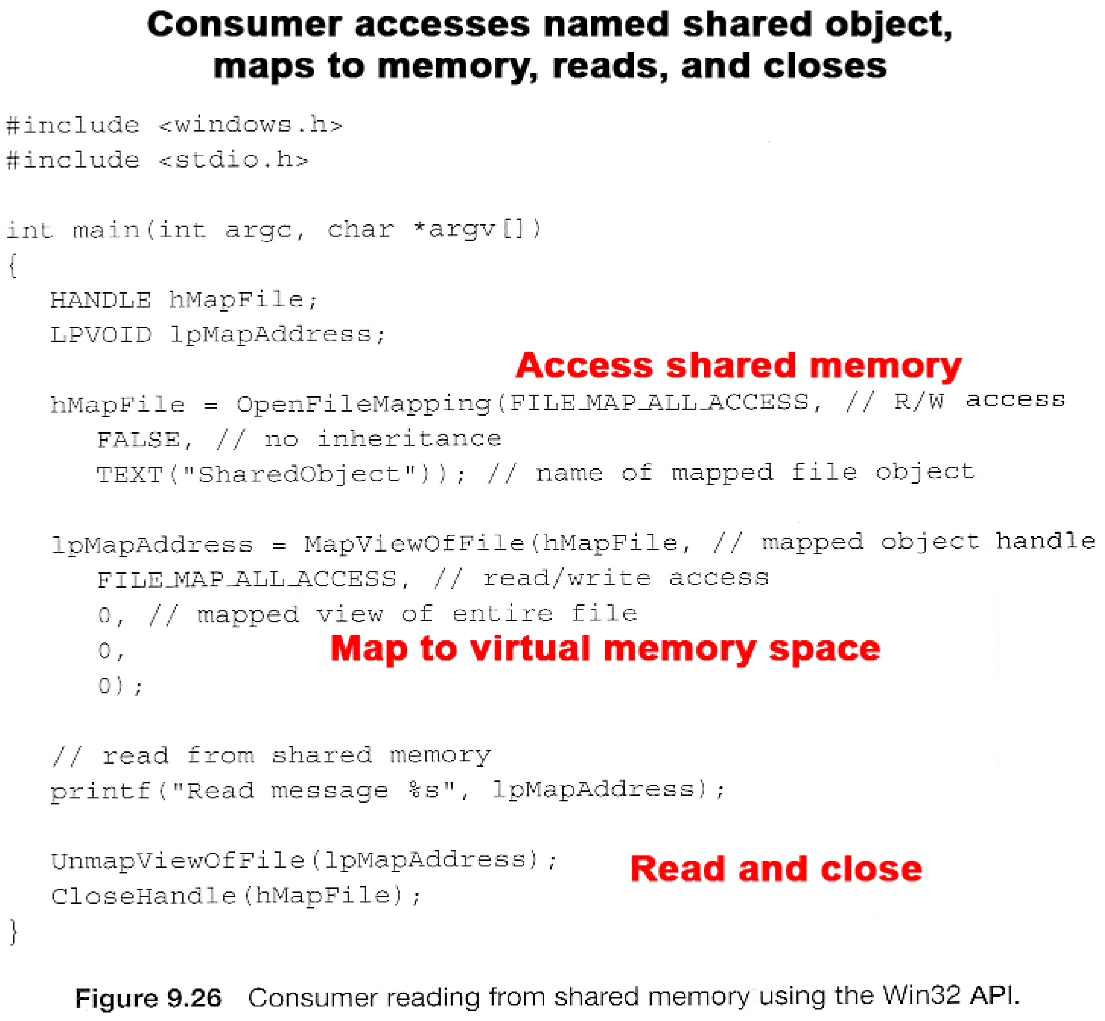
Make it a named shared object

Map it into virtual memory address space

Write and close

Figure 9.25 Producer writing to shared memory using the Win32 API.

Figure 9.24



Consumer accesses named shared object, maps to memory, reads, and closes

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   HANDLE hMapFile;
   LPVOID lpMapAddress;

   hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, // R/W access
      FALSE, // no inheritance
      TEXT("SharedObject")); // name of mapped file object

   lpMapAddress = MapViewOfFile(hMapFile, // mapped object handle
      FILE_MAP_ALL_ACCESS, // read/write access
      0, // mapped view of entire file
      0,
      0);

   // read from shared memory
   printf("Read message %s", lpMapAddress);

   UnmapViewOfFile(lpMapAddress);
   CloseHandle(hMapFile);
}
```

Access shared memory

Map to virtual memory space

Read and close

Figure 9.26 Consumer reading from shared memory using the Win32 API.

Figure 9.25

9.7.3 Memory-Mapped I/O

- All access to devices is done by writing into (or reading from) the device's registers. Normally this is done via special I/O instructions.
- For certain devices it makes sense to simply map the device's registers to addresses in the process's virtual address space, making device I/O as fast and simple as any other memory access. Video controller cards are a classic example of this.
- Serial and parallel devices can also use memory mapped I/O, mapping the device registers to specific memory addresses known as ***I/O Ports***, e.g. 0xF8. Transferring a series of bytes must be done one at a time, moving only as fast as the I/O device is prepared to process the data, through one of two mechanisms:
    - ***Programmed I/O (PIO)***, also known as ***polling.*** The CPU periodically checks the control bit on the device, to see if it is ready to handle another byte of data.
    - ***Interrupt Driven.*** The device generates an interrupt when it either has another byte of data to deliver or is ready to receive another byte.

9.8 Allocating Kernel Memory

- Previous discussions have centered on process memory, which can be conveniently broken up into page-sized chunks, and the only fragmentation that occurs is the average half-page lost to internal fragmentation for each process (segment.)
- There is also additional memory allocated to the kernel, however, which cannot be so easily paged. Some of it is used for I/O buffering and direct access by devices, example, and must therefore be contiguous and not affected by paging. Other memory is used for internal kernel data structures of various sizes, and since kernel memory is often locked (restricted from being ever swapped out), management of this resource must be done carefully to avoid internal fragmentation or other waste. (I.e. you would like the kernel to consume as little memory as possible, leaving as much as possible for user processes.) Accordingly there are several classic algorithms in place for allocating kernel memory structures.

9.8.1 Buddy System

- The Buddy ***System*** allocates memory using a ***power of two allocator***.
- Under this scheme, memory is always allocated as a power of 2 (4K, 8K, 16K, etc), rounding up to the next nearest power of two if necessary.
- If a block of the correct size is not currently available, then one is formed by splitting the next larger block in two, forming two matched buddies. (And if that larger size is not available, then the next largest available size is split, and so on.)
- One nice feature of the buddy system is that if the address of a block is exclusively ORed with the size of the block, the resulting address is the address of the buddy of the same size, which allows for fast and easy ***coalescing*** of free blocks back

into larger blocks.

  - Free lists are maintained for every size block.
  - If the necessary block size is not available upon request, a free block from the next largest size is split into two buddies of the desired size. (Recursively splitting larger size blocks if necessary.)
  - When a block is freed, its buddy's address is calculated, and the free list for that size block is checked to see if the buddy is also free. If it is, then the two buddies are coalesced into one larger free block, and the process is repeated with successively larger free lists.
  - See the (annotated) Figure 9.27 below for an example.

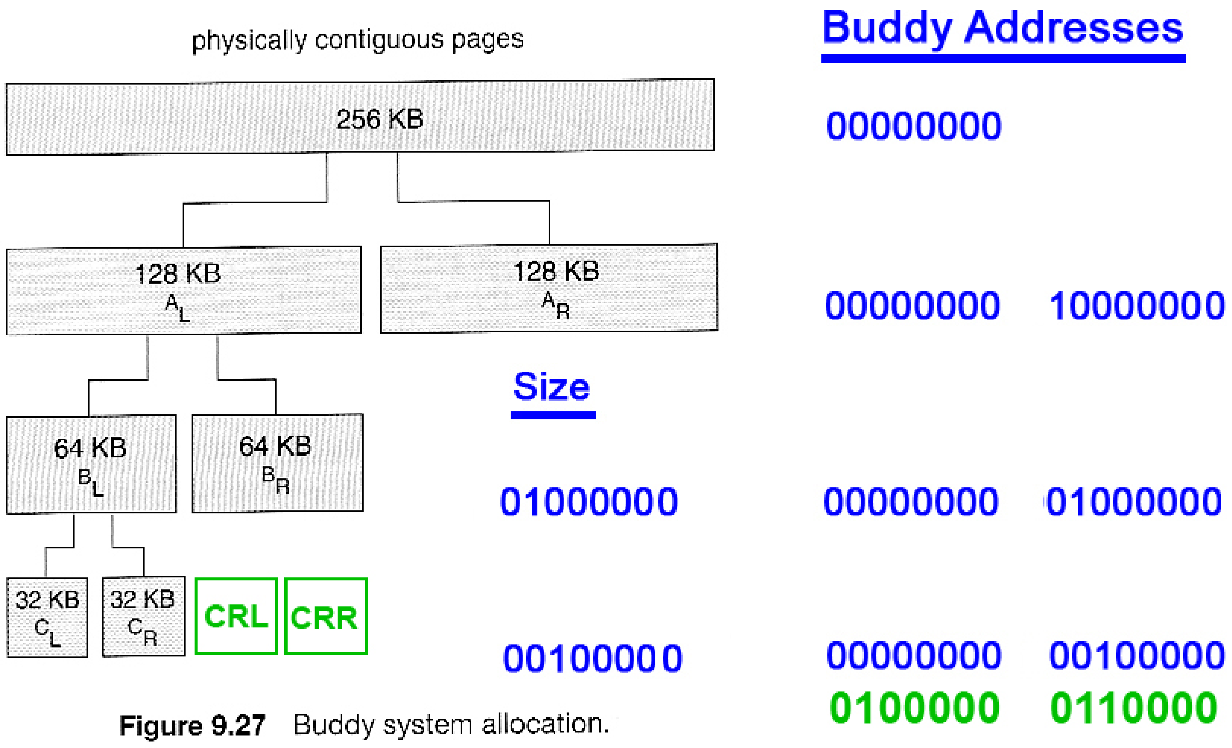


Figure 9.27 Buddy system allocation.

Figure 9.26

9.8.2 Slab Allocation

- ***Slab Allocation*** allocates memory to the kernel in chunks called ***slabs***, consisting of one or more contiguous pages. The kernel then creates separate caches for each type of data structure it might need from one or more slabs. Initially the caches are marked empty, and are marked full as they are used.
- New requests for space in the cache is first granted from empty or partially empty slabs, and if all slabs are full, then additional slabs are allocated.
- (This essentially amounts to allocating space for arrays of structures, in large chunks suitable to the size of the structure being stored. For example if a particular structure were 512 bytes long, space for them would be allocated in groups of 8 using 4K pages. If the structure were 3K, then space for 4 of them could be allocated at one time in a slab of 12K using three 4K pages.
- Benefits of slab allocation include lack of internal fragmentation and fast allocation of space for individual structures
- Solaris uses slab allocation for the kernel and also for certain user-mode memory allocations. Linux used the buddy system prior to 2.2 and switched to slab allocation since then.
- **New in 9th Edition: Linux SLOB and SLUB allocators replace SLAB**
  - SLOB, Simple List of Blocks, maintains 3 linked lists of free blocks - small, medium, and large - designed for (imbedded) systems with limited amounts of memory.
  - SLUB modifies some implementation issues for better performance on systems with large numbers of processors.

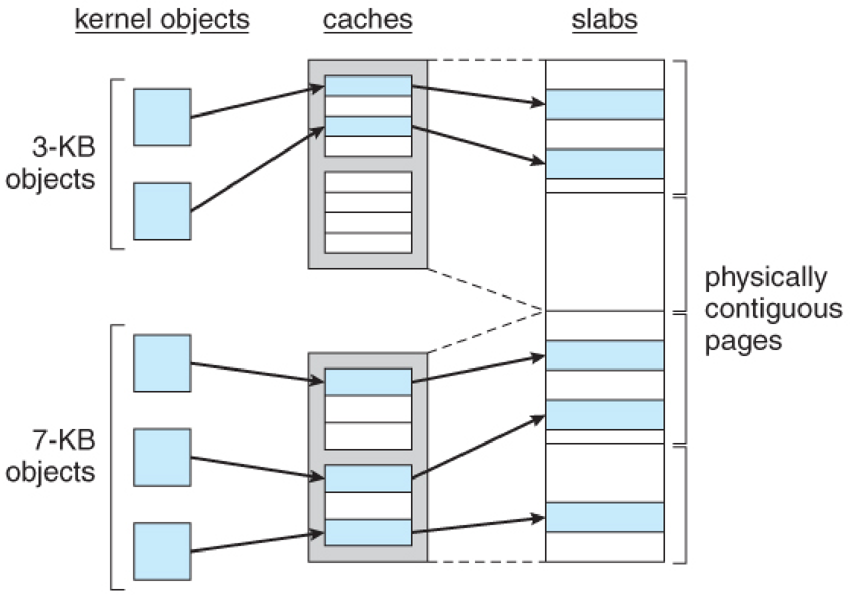


Figure 9.27 - Slab allocation.

9.9 Other Considerations

9.9.1 Prepaging

- The basic idea behind ***prepaging*** is to predict the pages that will be needed in the near future, and page them in before they are actually requested.
- If a process was swapped out and we know what its working set was at the time, then when we swap it back in we can go ahead and page back in the entire working set, before the page faults actually occur.
- With small (data) files we can go ahead and prepage all of the pages at one time.
- Prepaging can be of benefit if the prediction is good and the pages are needed eventually, but slows the system down if the prediction is wrong.

9.9.2 Page Size

- There are quite a few trade-offs of small versus large page sizes:
- Small pages waste less memory due to internal fragmentation.
- Large pages require smaller page tables.
- For disk access, the latency and seek times greatly outweigh the actual data transfer times. This makes it much faster to transfer one large page of data than two or more smaller pages containing the same amount of data.
- Smaller pages match locality better, because we are not bringing in data that is not really needed.
- Small pages generate more page faults, with attending overhead.
- The physical hardware may also play a part in determining page size.
- It is hard to determine an "optimal" page size for any given system. Current norms range from 4K to 4M, and tend towards larger page sizes as time passes.

9.9.3 TLB Reach

- ***TLB Reach*** is defined as the amount of memory that can be reached by the pages listed in the TLB.
- Ideally the working set would fit within the reach of the TLB.
- Increasing the size of the TLB is an obvious way of increasing TLB reach, but TLB memory is very expensive and also draws lots of power.
- Increasing page sizes increases TLB reach, but also leads to increased fragmentation loss.
- Some systems provide multiple size pages to increase TLB reach while keeping fragmentation low.
- Multiple page sizes requires that the TLB be managed by software, not hardware.

9.9.4 Inverted Page Tables

- Inverted page tables store one entry for each frame instead of one entry for each virtual page. This reduces the memory requirement for the page table, but loses the information needed to implement virtual memory paging.
- A solution is to keep a separate page table for each process, for virtual memory management purposes. These are kept on disk, and only paged in when a page fault occurs. (I.e. they are not referenced with every memory access the way a traditional page table would be.)

9.9.5 Program Structure

- Consider a pair of nested loops to access every element in a 1024 x 1024 two-dimensional array of 32-bit ints.
- Arrays in C are stored in row-major order, which means that each row of the array would occupy a page of memory.
- If the loops are nested so that the outer loop increments the row and the inner loop increments the column, then an entire row can be processed before the next page fault, yielding 1024 page faults total.

- On the other hand, if the loops are nested the other way, so that the program worked down the columns instead of across the rows, then every access would be to a different page, yielding a new page fault for each access, or over a million page faults all together.
- Be aware that different languages store their arrays differently. FORTRAN for example stores arrays in column-major format instead of row-major. This means that blind translation of code from one language to another may turn a fast program into a very slow one, strictly because of the extra page faults.

9.9.6 I/O Interlock and Page Locking

There are several occasions when it may be desirable to ***lock*** pages in memory, and not let them get paged out:

- Certain kernel operations cannot tolerate having their pages swapped out.
- If an I/O controller is doing direct-memory access, it would be wrong to change pages in the middle of the I/O operation.
- In a priority based scheduling system, low priority jobs may need to wait quite a while before getting their turn on the CPU, and there is a danger of their pages being paged out before they get a chance to use them even once after paging them in. In this situation pages may be locked when they are paged in, until the process that requested them gets at least one turn in the CPU.

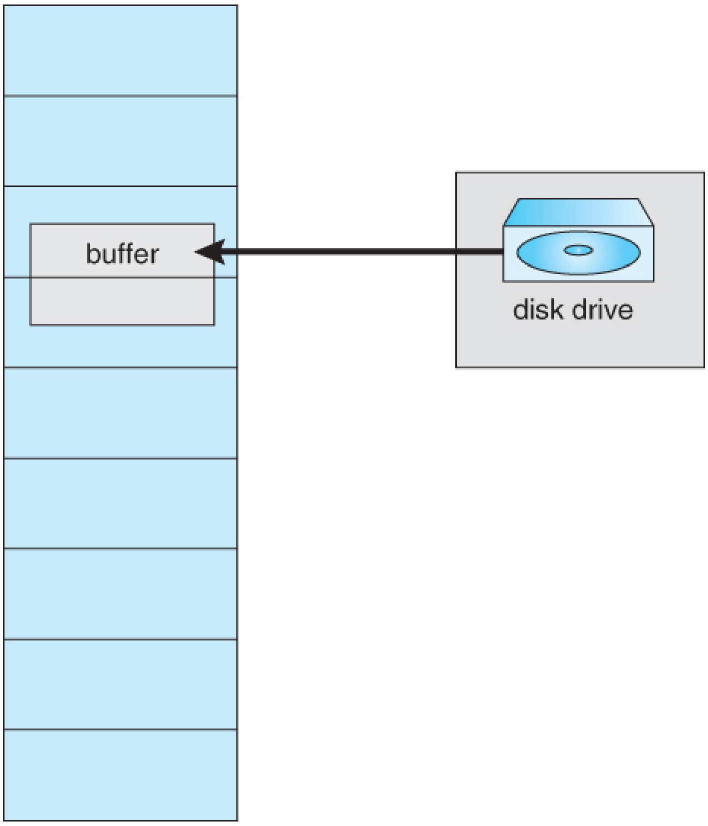


Figure 9.28 - The reason why frames used for I/O must be in memory.

9.10 Operating-System Examples (Optional)

9.10.1 Windows

- Windows uses demand paging with ***clustering,*** meaning they page in multiple pages whenever a page fault occurs.
- The working set minimum and maximum are normally set at 50 and 345 pages respectively. (Maximums can be exceeded in rare circumstances.)
- Free pages are maintained on a free list, with a minimum threshold indicating when there are enough free frames available.
- If a page fault occurs and the process is below their maximum, then additional pages are allocated. Otherwise some pages from this process must be replaced, using a local page replacement algorithm.
- If the amount of free frames falls below the allowable threshold, then ***working set trimming*** occurs, taking frames away from any processes which are above their minimum, until all are at their minimums. Then additional frames can be allocated to processes that need them.
- The algorithm for selecting victim frames depends on the type of processor:
  - On single processor 80x86 systems, a variation of the clock (second chance) algorithm is used.
  - On Alpha and multiprocessor systems, clearing the reference bits may require invalidating entries in the TLB on other processors, which is an expensive operation. In this case Windows uses a variation of FIFO.

9.10.2 Solaris

- Solaris maintains a list of free pages, and allocates one to a faulting thread whenever a fault occurs. It is therefore imperative that a minimum amount of free memory be kept on hand at all times.
- Solaris has a parameter, *lotsfree,* usually set at 1/64 of total physical memory. Solaris checks 4 times per second to see if the free memory falls below this threshhold, and if it does, then the ***pageout*** process is started.

- Pageout uses a variation of the clock (second chance) algorithm, with two hands rotating around through the frame table. The first hand clears the reference bits, and the second hand comes by afterwards and checks them. Any frame whose reference bit has not been reset before the second hand gets there gets paged out.
- The Pageout method is adjustable by the distance between the two hands, (the ***handspan***), and the speed at which the hands move. For example, if the hands each check 100 frames per second, and the handspan is 1000 frames, then there would be a 10 second interval between the time when the leading hand clears the reference bits and the time when the trailing hand checks them.
- The speed of the hands is usually adjusted according to the amount of free memory, as shown below. ***Slowscan*** is usually set at 100 pages per second, and ***fastscan*** is usually set at the smaller of 1/2 of the total physical pages per second and 8192 pages per second.

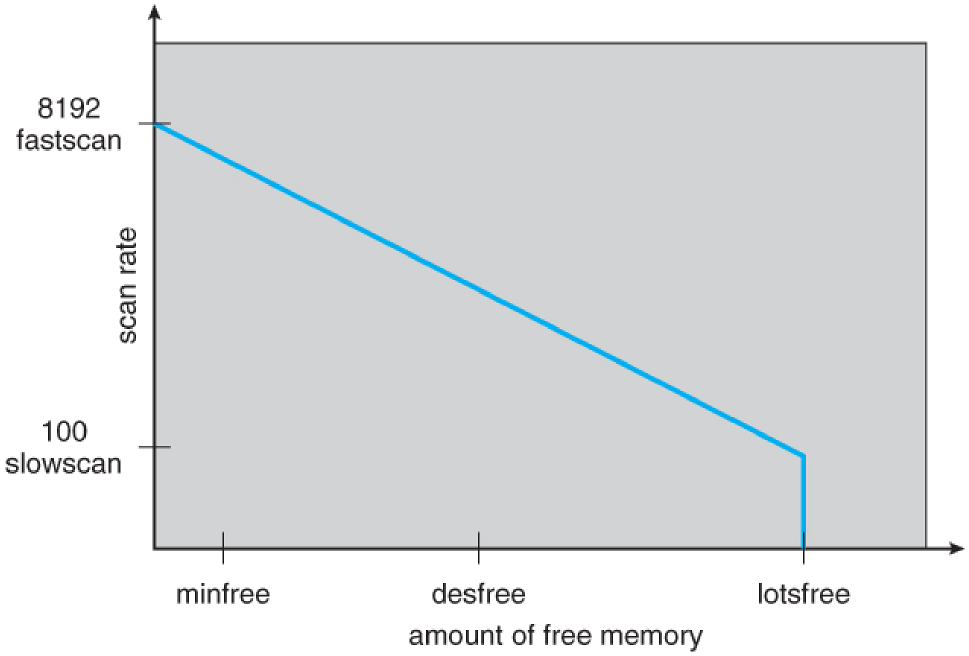


Figure 9.29 - Solaris page scanner.

- Solaris also maintains a cache of pages that have been reclaimed but which have not yet been overwritten, as opposed to the free list which only holds pages whose current contents are invalid. If one of the pages from the cache is needed before it gets moved to the free list, then it can be quickly recovered.
- Normally pageout runs 4 times per second to check if memory has fallen below *lotsfree*. However if it falls below *desfree,* then pageout will run at 100 times per second in an attempt to keep at least desfree pages free. If it is unable to do this for a 30-second average, then Solaris begins swapping processes, starting preferably with processes that have been idle for a long time.
- If free memory falls below *minfree,* then pageout runs with every page fault.
- Recent releases of Solaris have enhanced the virtual memory management system, including recognizing pages from shared libraries, and protecting them from being paged out.

9.11 Summary