Module IV
Virtual-Memory Management

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.

- 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. Certain features of certain programs are rarely used.
- The ability to load only the portions of processes that are actually 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.

The figure below shows the general layout of ***virtual memory***, which can be much larger than physical memory:

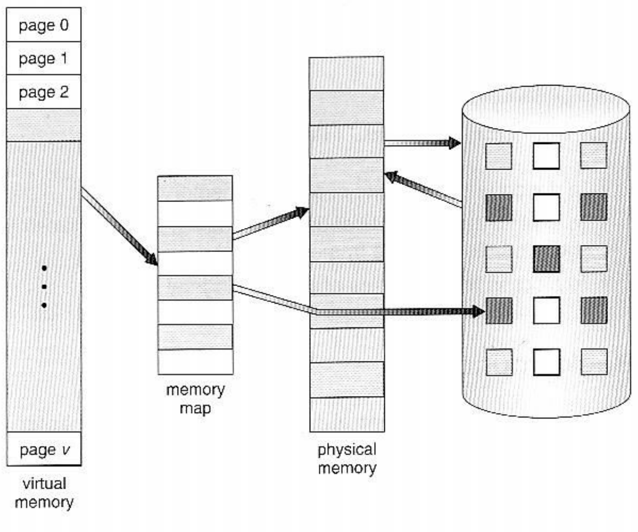

Figure 9.1 Diagram showing virtual memory that is larger than physical memory

- The figure below shows ***virtual address space***, which is the programmer's 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.

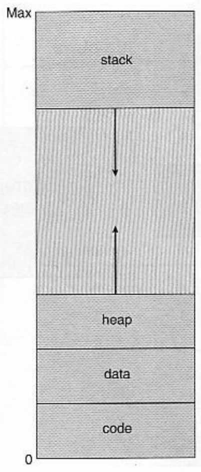

Figure 9.2 Virtual address space.

- 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.

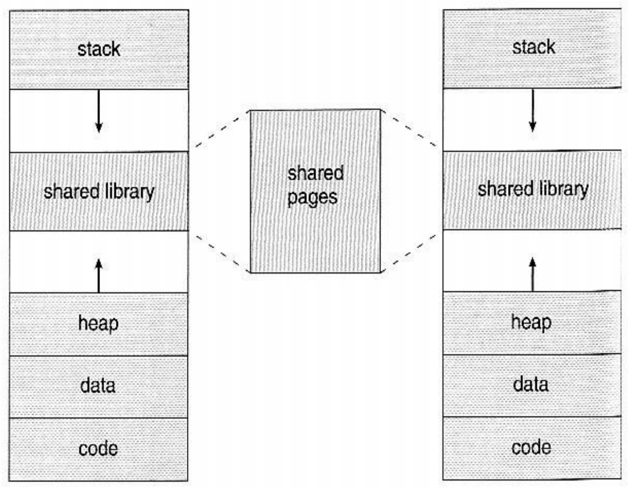

Figure 9.3 Shared library using virtual memory.

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 as ***lazy swapper***, although a ***pager*** is a more accurate term.

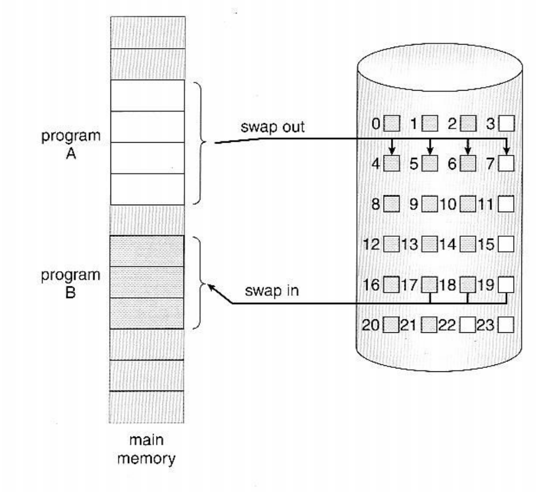

Figure 9.4 Transfer of a paged memory to contiguous disk space.

- The basic idea behind demand paging is that when a process is swapped in, the pager only loads into memory those pages that is needed presently.
- Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit. Pages loaded in memory are marked as valid.
- 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.

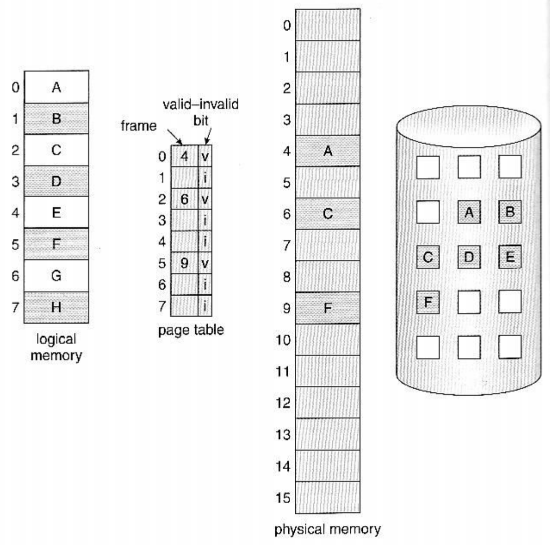

Figure 9.5 Page table when some pages are not in main memory.

- 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 is to an invalid page, the process is terminated. Otherwise, if the page is not present in memory, it 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.
    5. After the page is loaded to memory, 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.

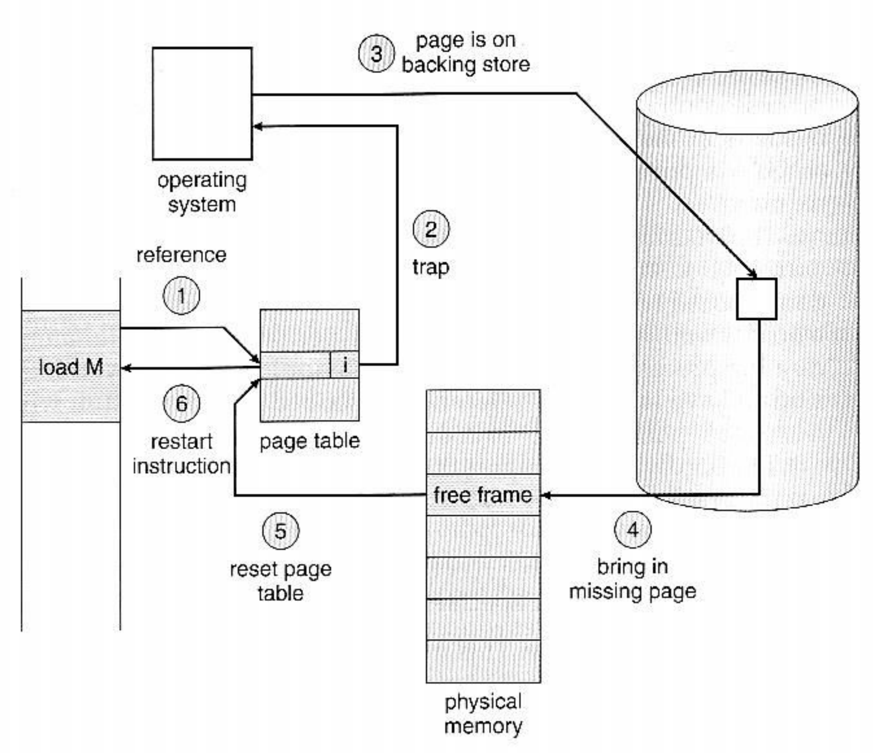

Figure 9.6 Steps in handling a page fault.

- In an extreme case, the program starts execution with zero pages in memory. Here NO pages are swapped in for a process until they are requested by page faults. This is known as ***pure demand paging.***
- The **hardware necessary** to support demand paging is the same as for paging and swapping: A page table and secondary memory.

Performance of Demand Paging

- There is some slowdown and performance hit whenever a page fault occurs(as the required page is not available in memory) and the system has to go get it from memory.
- There are many steps that occur when servicing a page fault 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:

Effective access time = p * time taken to access memory in page fault+ (1-p)* time taken to access memory

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

= 200 + 7,999,800 * 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.

9.3 Copy-on-Write

- The idea behind a copy-on-write is that the pages of a parent process is shared by the child process, until one or the other of the processes changes the page. Only when a process changes any page content, that page is copied for the child.

- Only pages that can be modified need to be labeled as copy-on-write. Code segments can simply be shared.
- 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

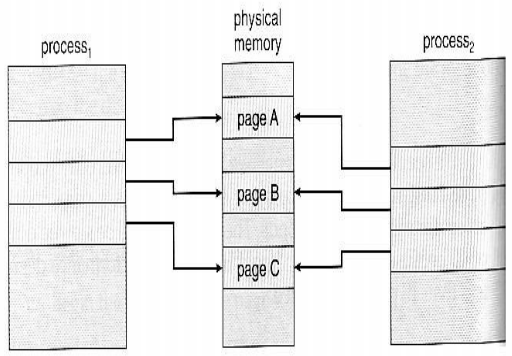

Figure 9.7 Before process 1 modifies page C.

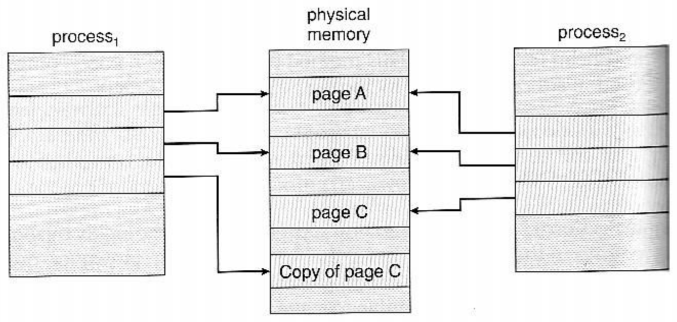

Figure 9.8 After process 1 modifies page C.

before performing the exec( ) system call.

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 are frames to load many more processes in memory.
- If some process suddenly decides to use more pages and there aren't any free frames available. Then 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.
    2. Put the process requesting more pages into a <u>wait queue</u> until some free frames become available.
    3. Swap some process out of memory <u>completely</u>, 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.

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.
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. Change the entries in page table.
4. Restart the process that was waiting for this page.

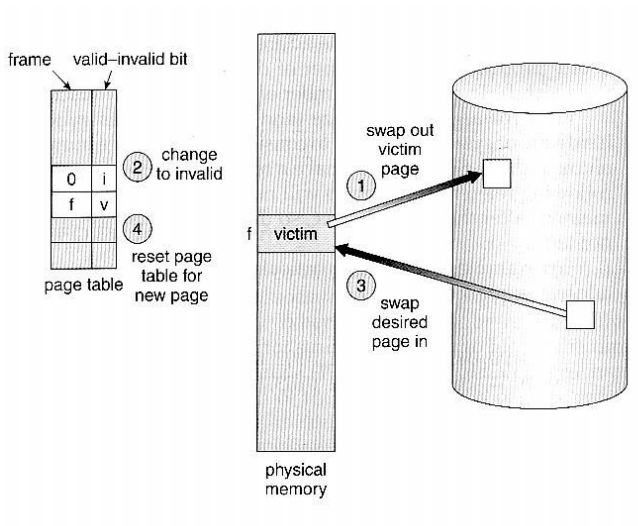

Figure 9.10 Page replacement.

- Note that step 2c adds an extra disk write to the page-fault handling, thus doubling the time required to process a page fault. This can be reduced by assigning a ***modify bit,*** or ***dirty bit*** to each page in memory, indicating whether or not it has been changed since it was last loaded in from disk. If the page is not modified the bit is not set. If the dirty bit has not been set, then the page is unchanged, and does not need to be written out to disk. Many page replacement strategies specifically look for pages that do not have their dirty bit set.
- There are two major requirements to implement a successful demand paging system.

  A ***frame-allocation algorithm*** and a ***page-replacement algorithm.*** The former centers around how many frames are allocated to each process, 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 page accesses known as a ***reference string.***

Few Page Replacement algorithms –

a) FIFO Page Replacement

- A simple and obvious page replacement strategy is ***FIFO***, i.e. first-in-first-out.
- This algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.

- Or a FIFO queue can be created to hold all pages in memory. 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, a reference string is given and there are 3 free frames. There are 20 page requests, which results in 15 page faults.

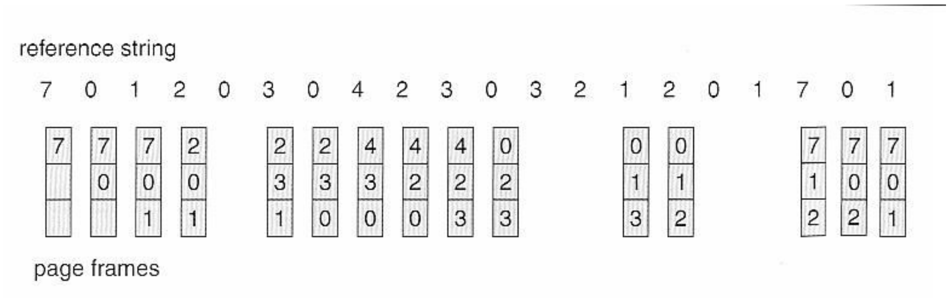

Figure 9.12 FIFO page-replacement algorithm.

- Although FIFO is simple and easy to understand, it is not always optimal, or even efficient.
- ***Belady's anomaly*** tells that for some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases.

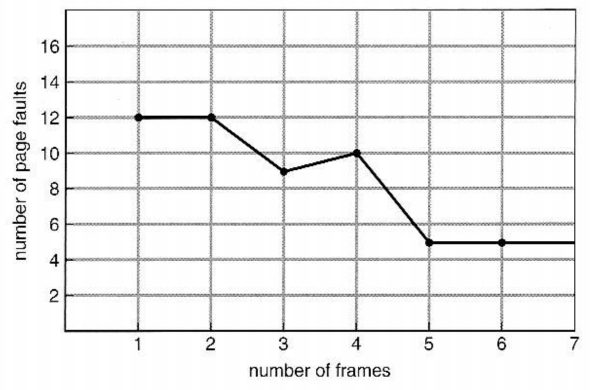

Figure 9.13 Page-fault curve for FIFO replacement on a reference string.

b) 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 "Replace the page that will not be used for the longest time in the future."
- The same reference string used for the FIFO example is used in the example below, here the minimum number of possible page faults is 9.

- Unfortunately OPT cannot be implemented in practice, because it requires the knowledge of future string, but it makes a nice benchmark for the comparison and evaluation of real proposed new algorithms.

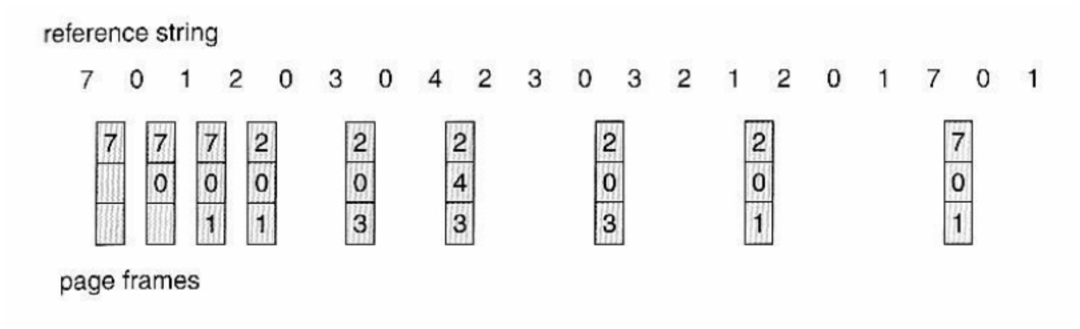

Figure 9.14 Optimal page-replacement algorithm.

- LRU is considered a good replacement policy, and is often used. There are two simple approaches commonly used to implement this:
    1. **Counters.** With each page-table entry a time-of-use field is associated. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. This scheme requires a search of the page table to find the LRU page and a write to memory for each memory access.

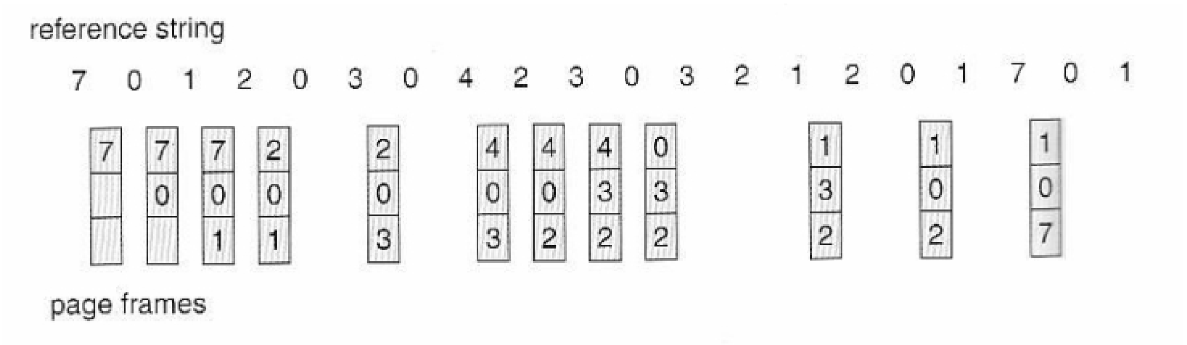

Figure 9.15 LRU page-replacement algorithm.

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.

- 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.

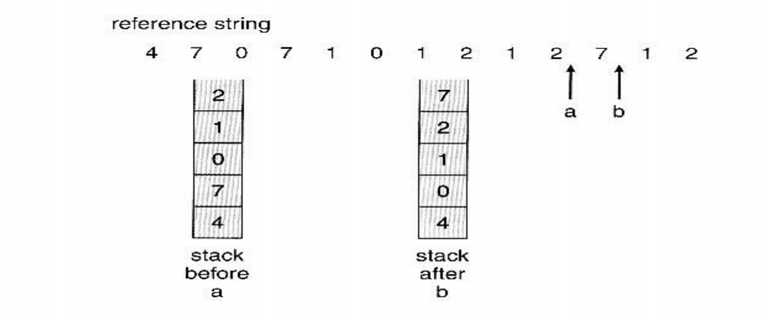

Figure 9.16 Use of a stack to record the most recent page references.

d) LRU-Approximation Page Replacement

- Many systems offer some degree of hardware support, enough to approximate LRU.
- 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 distinguishes pages that have been accessed since the last clear from those that have not been accessed.

d.1 Additional-Reference-Bits Algorithm

- An 8-bit byte(reference bit) is stored for each page in a table in memory.
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
- These 8-bit shift registers contain the history of page use for the last eight time periods.
- If the shift register contains 00000000, then the page has not been used for eight time periods.
- A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.

d.2 Second-Chance Algorithm

- The ***second chance algorithm*** is a FIFO replacement algorithm, 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 as '0', then that page is selected as the next victim.
- If the reference bit value is '1', then the page is given a second chance and its reference bit value is cleared(assigned as'0').

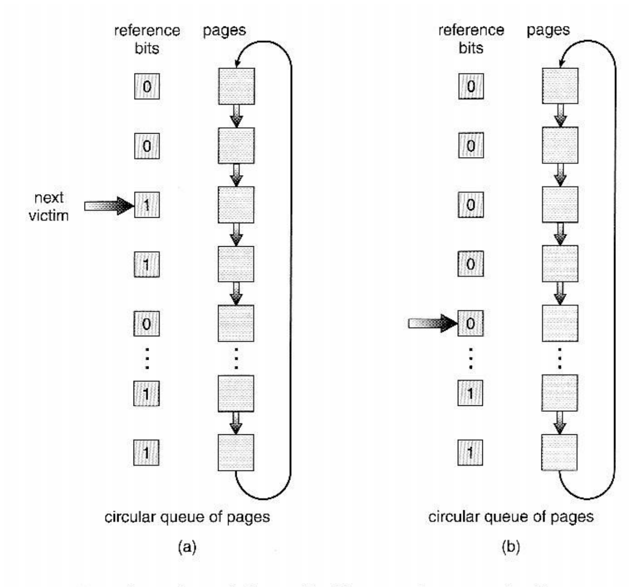

Figure 9.17 Second-chance (clock) page-replacement algorithm.

- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often, then it sets its reference bit again.
- This algorithm is also known as the ***clock*** algorithm.

One way to implement the second-chance algorithm is as a circular queue. *A* pointer indicates which page is *to* be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits. Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.

d.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, 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.

e) 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.

f) Page-Buffering 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, so that the requesting process is in memory as early as possible, and then select a victim page to write to disk and free up a frame.
- Keep a list of modified pages, and when the I/O system is idle, these pages are written to disk, and then clear the modify bits, thereby increasing the chance of finding a "clean" page for the next potential victim and page replacement can be done much faster.

9.5 Allocation of Frames

- The absolute minimum number of frames that a process must be allocated is dependent on system architecture.
- The maximum number is defined by the amount of available physical memory.

Allocation Algorithms

After loading of OS, there are two ways in which the allocation of frames can be done to the processes.

- **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 depending on the size of the process. If the size of process i is S_i, and S is the sum of size of all processes in the system, then the allocation for process P_i is $a_i = m * S_i / S$. where m is the free frames available in the system.

Consider a system with a 1KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames.

With proportional allocation, we would split 62 frames between two processes, as follows-

m=62, S = (10+127)=137

Allocation for process 1 = 62 X 10/137 ~ 4

Allocation for process 2 = 62 X 127/137 ~57

Thus allocates 4 frames and 57 frames to student process and database respectively.

- Variations on proportional allocation could consider priority of process rather than just their size.

Global versus Local Allocation

- Page replacement can occur both at 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.

Non-Uniform Memory Access (New)

- Usually the time required to access all memory in a system is equivalent.
- 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 such 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.

9.6 Thrashing

Thrashing is the state of a process where there is **high paging activity**. A process that is spending more time paging than executing is said to be ***thrashing.***

9.6.1 Cause of Thrashing

- When memory is filled up and processes starts spending lots of time waiting for their pages to page in, then CPU utilization decreases(Processes are not executed as they are waiting for some pages), causing the scheduler to add in even more processes and increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. No work is getting done, because the processes are spending all their time paging.
- In the graph given below , CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.

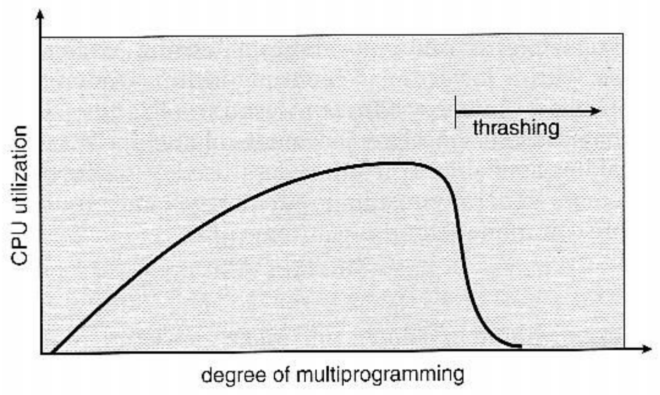

Figure 9.18 Thrashing.

- Local page replacement policies can prevent thrashing process from taking pages away from other processes, but it still tends to clog up the I/O queue.

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:

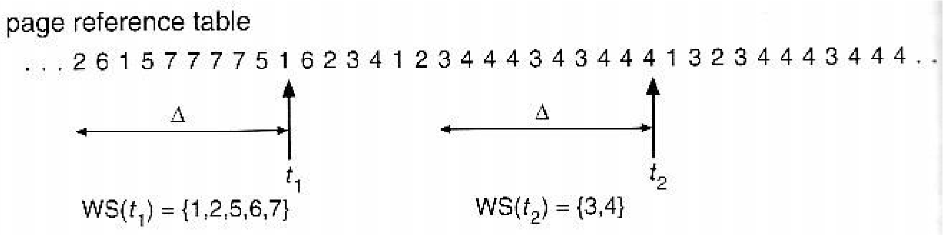

Figure 9.20 Working-set model.

- 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 of frames, D, is the sum of the sizes of the working sets for all processes (D=WSS$_i$). 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.

9.6.3 Page-Fault Frequency

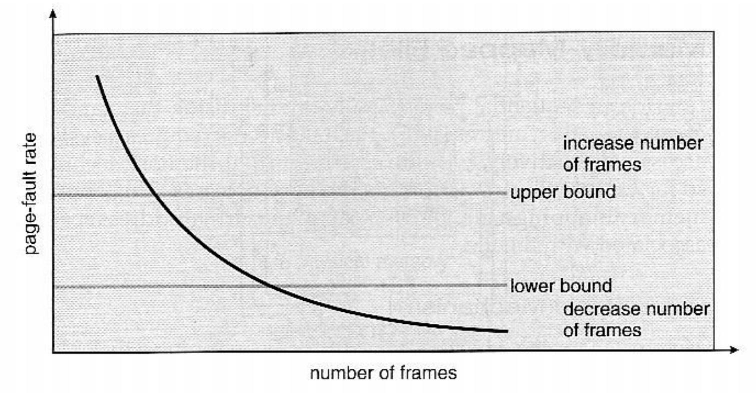

Figure 9.21 Page-fault frequency.

- When page-fault rate is too high, the process needs more frames and when it is too low, the process may have too many frames.
- The upper and lower bounds can be established on the page-fault rate. If the actual page-fault rate exceeds the upper limit, allocate the process another frame or suspend the process. If the page-fault rate falls below the lower limit, remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.