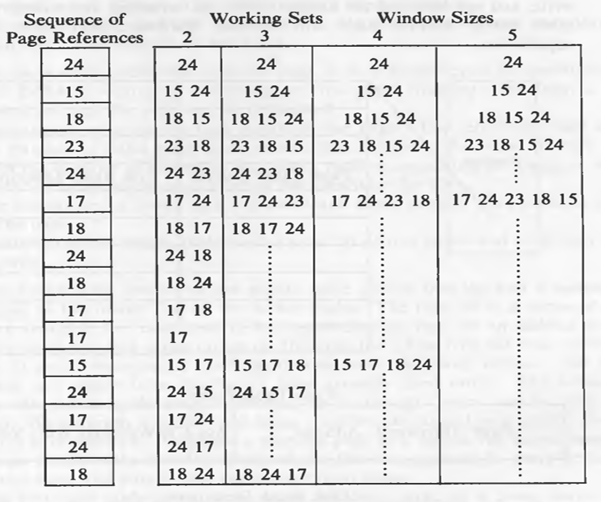
**1. What is demand paging? Explain data structure used for demand paging?**

**7. Explain the data structures for demand paging?**

* Demand paging can be described as a memory management technique that is used in operating systems to improve memory usage and system performance. Demand paging is a technique used in virtual memory systems where pages enter main memory only when requested or needed by the CPU.
* In demand paging, the operating system loads only the necessary pages of a program into memory at runtime, instead of loading the entire program into memory at the start.
* A page fault occurred when the program needed to access a page that is not currently in memory. The operating system then loads the required pages from the disk into memory and updates the page tables accordingly. This process is transparent to the running program and it continues to run as if the page had always been in memory.
* The portion of the text section which contains the frequently called subroutines and program loops is a small subset of the total space of the process. This is known as the principle of "locality".
* The working set of a process is the set of pages that the process has referenced in its last n memory references; the number n is called the window of the working set.
* If the working set stays in the memory, swapping could be reduced. When a process addresses a page that is not in its working set, it incurs a page fault; in handling the fault, the kernel updates the working set, reading in pages from a secondary device if necessary.

The following figure shows the changes in working set of different window sizes as the process accesses

pages:



As shown in the diagram, working sets with larger window sizes are more efficient, implying that the

process will not fault often as often.

**Data Structures for Demand Paging**

The kernel contains 4 major data structures to support low-level memory management functions and demand

paging:

* page table entries
* disk block descriptors
* page frame data table (called pfdata for short)
* swap-use table

pfdata table is allocated once for the lifetime of the system, but allocations for other structure are dynamic.

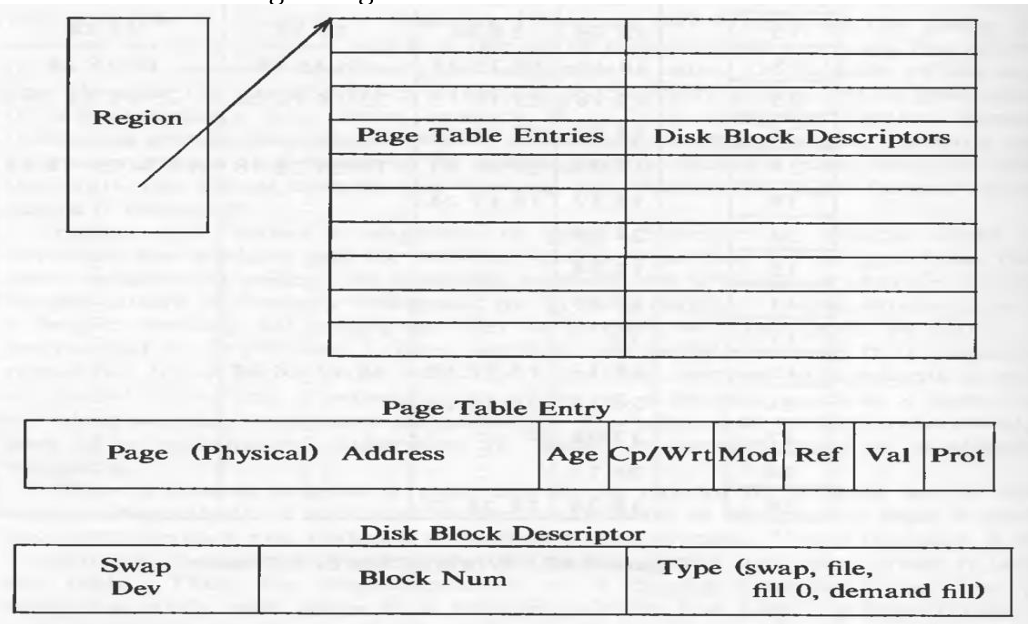
As seen previously, the region contains page tables to access physical memory. Each entry of a page table

contains the physical address of the page, protection bits indicating whether processes can read, write or

execute from the page, and the following bit fields to support demand paging:

* Valid
* Reference
* Modify
* Copy on write
* Age
* The kernel sets the valid bit to indicate that the contents of a page are legal, but the page reference is not necessarily illegal if the valid bit is not set.
* The reference bit indicates whether a process recently referenced a page,
* And the modify bit indicates whether a process recently modified the contents of a page.
* The copy on write bit, used in the fork system call, indicates that the kernel must create a new copy of the page when a process modifies its contents.
* The kernel manipulates the age bits to indicate how long a page has been a member of the working set of a process.

The structure of the region is given below:



Each page table entry is associated with a disk block descriptor, which describes the disk copy of the virtual

page. Processes that share a region therefore access common page table entries and disk block descriptors.

The contents of a virtual pages are either in a particular block on a swap device, in an executable, or not on a

swap device. If the page is on a swap device, the disk block descriptor contains the logical device number

and block number containing the page contents. If the page is contained in an executable file, the disk block

descriptor contains the logical block number in the file that contains the page; the kernel can quickly map

this number into its disk address. The disk block descriptor also indicates two special conditions set

during exec: that a page is "demand fill" or "demand zero", studied later.

The pfdata table describes each page of physical memory and is indexed by page number. It contains these

fields:

* State of the page, indicating that the page is on a swap device or executable file, that DMA is currently underway for the page (reading data from a swap device), or that the page can be reassigned.
* The number of processes that reference the page. The reference count equals the number of valid page table entries that reference the page. It may differ from the number of processes that share regions containing the page, as will be described below when reconsidering the algorithm fork.
* The logical device (swap or file system) and block number that contains a copy of the page.
* Pointers to other pfdata table entries on a list of free pages and on a hash queue of pages.

The kernel links entries of the pfdata table onto a free list and a hashed list, just like the buffer cache. This

free list is also used in LRU order. The hash queues are hashes according to the (swap) device number and

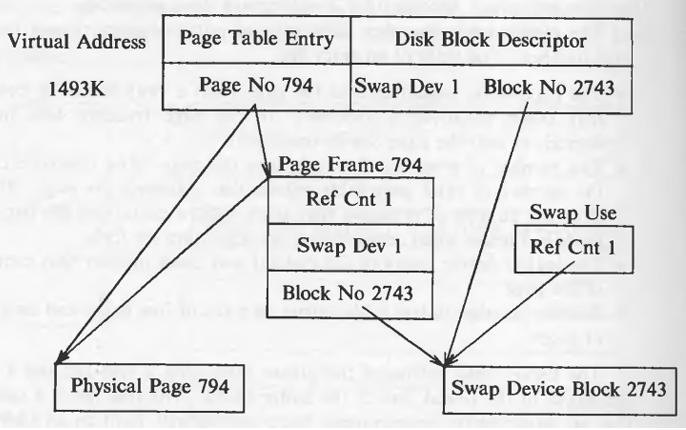
block number.

The swap-use table contains an entry for every page on a swap device. The entry consists of a reference

count of how many page table entries point to a page on a swap device.

The following figure shows the relationship between page table entries, disk block descriptors, pfdata table

entries, and the swap-use count table.



**Advantages of Demand Paging**

1. Optimal Memory Utilization
2. Fast Process
3. Virtual Memory Support
4. Improved System Responsiveness
5. Efficient Multi programming

**Disadvantages of Demand Paging**

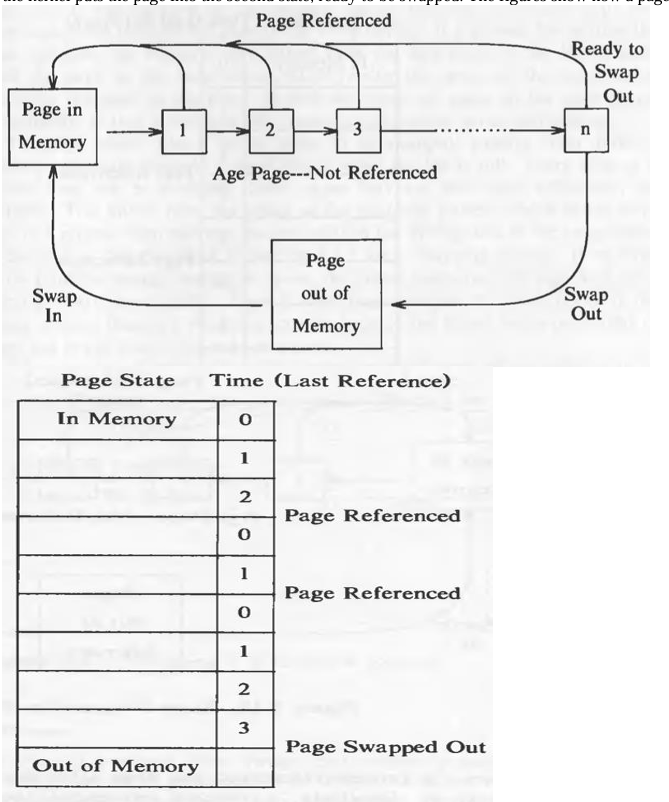
1. Demand paging incurs overhead in terms of time and system resources. When a page fault occurs (that is, a necessary page is not in physical memory), the operating system must recover the page from secondary storage (for example, a hard drive or SSD). This method involves disk I/O, which is slower than reading data from RAM. This might cause latency and lower overall system performance.
2. Handling page faults can be difficult and resource-intensive for the operating system. The system must locate the appropriate page on secondary storage, load it into RAM, and update the page tables. Frequent page errors can result in unnecessary overhead, especially in settings with limited physical memory.
3. Garbage occurs when the system spends more time going in and out of physical memory rather than completing important tasks. It may be caused by insufficient physical memory to meet the demands of the operating system. Turbulence can slow down the body’s performance and make it unresponsive.
4. Disk fragmentation can occur when pages are frequently moved in and out of secondary storage. This fragmentation can cause additional delays when accessing data because required pages may be scattered across the disk rather than contiguous.

2. Explain the working of page stealer process.

The Page-Stealer Process:

* The page stealer is a kernel process that swaps out memory pages that are no longer part of the working set of a process.
* It is created on system initialization and is invoked throughout the system lifetime when system is low on free pages.
* It examines every active, unlocked region and increments the age field of all valid pages. The kernel locks a region when a process faults on a page in the region, so that the page stealer cannot steal the page being faulted in.
* There are two paging states for a page in memory: the page is aging and is not yet eligible for swapping, or the page is eligible for swapping and is available for reassignment to other virtual pages.
* The first state indicates that a process recently accessed the page. Some machines set a reference bit when they reference a page, but software methods can be substituted if the hardware does not have this feature. The page stealer turns off the reference bit for such pages but remembers how many examinations have passed since the page was last referenced. The first state thus consists of several substates, corresponding to the number of passes the page stealer makes before the page is eligible for swapping. When the number exceeds a threshold value, the kernel puts the page into the second state, ready to be swapped.

The figures show how a page ages:



Because of sharing, a page can be part of the working set of one or more processes. But this does not matter

to the page stealer. It is just that there is more chance of the page getting referred. The kernel wakes up the page stealer when memory is below a low-water mark and the page stealer swaps out pages until the

available memory goes above a high-water mark. Because of swapping out up till a high-water mark, the

memory usage does not go below to the low-water mark quickly. Administrators can configure the low and

high water mark values.

**When the page stealer decides to swap out a page, it considers whether a copy of the page is on a swap**

**device. There are three possibilities:**

* 1. If no copy of the page is on a swap device, the kernel "schedules" the page for swapping: The page stealer places the page on a list of pages to be swapped out and continues; the swap is logically complete. When the list of pages to be swapped reaches a limit (dependent on the capabilities of the disk controller), the kernel writes the pages to the swap device.
  2. If a copy of the page is already on a swap device and no process had modified its in-core contents (the page table entry modify bit is clear), the kernel clears the page table entry valid bit, decrements the reference count in the pfdata table entry, and puts the entry on the free list for future allocation.
  3. If a copy of a page is on a swap device but the contents are modified in memory, the kernel schedules the page for swapping, as above, and frees the space it currently occupies on the swap device.

The page stealer fills a list of pages to be swapped and swaps them to a swap device when the list is full. If

no swap device contains enough contiguous space, the kernel swaps out one page at a time (this is more costly). There is more fragmentation of a swap device in the paging scheme than in a swapping scheme,

because the kernel swaps out block of pages but swaps in only one page at time.

When the kernel writes a page to swap device, it turns off the valid bit in its page table entry and decrements

the use of count of its pfdata table entry. If the count drops to 0, it places the pfdata table entry at the end of

the free list, caching it until reassignment. If the count is not 0, several processes are sharing the page as a

result of a previous fork call, but the kernel still swaps the page out. Finally, the kernel allocates swap space,

saves the swap address in the disk block descriptor, and increments the swap-use table count for the page. If

a process incurs a page fault while the page is on the free list, however, the kernel can rescue the page from

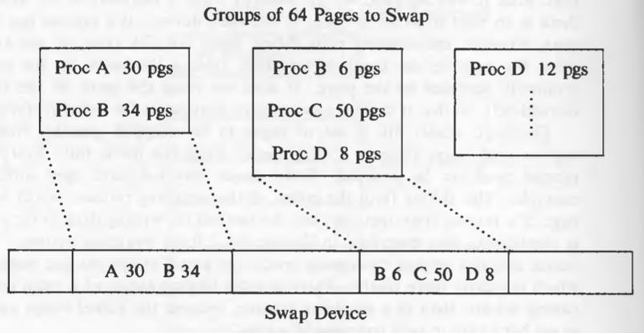
memory instead of having to retrieve it form the swap space. However, the page is still swapped if it is on

the swap lists.

For example, suppose the page stealer swaps out 30, 40, 50, and 20 pages from processes A, B, C, and D,

respectively, and that it writes 64 pages to the swap device in one disk write operation. The figure shows the

sequence of page-swapping operations that would occur if the order is A, B, C, and D.



3. What is page fault? Explain handling of validity page fault.

There are two types of page faults: validity faults and protection faults. The fault handlers may have to read

a page from disk to memory and sleep during the I/O operation, therefore, they are an exception to the

general rule that interrupt handlers cannot sleep.

**Validity Fault Handler**

If a process attempts to access a page whose valid bit is not set, it incurs a validity fault and then the kernel

invokes the validity fault handler. The algorithm vfault is given below:

/\* Algorithm: vfault

\* Input: address where process faulted

\* Output: none

\*/

{

find region, page table entry, disk block descriptor corresponding to faulted address, lock region;

if (address outside virtual address space)

{

send signal (SIGSEGV: segmentation violation) to process;

goto out;

}

if (address now valid) // process may have slept above

goto out;

if (page in cache)

{

remove page from cache;

adjust page table entry;

while (page contents not valid) // another proc faulted first

sleep (event: contents become valid);

}

else

{

assign new page to region;

put new page in cache, update pfdata entry;

if (page not previously loaded and page "demand zero")

clear assigned page to 0;

else

{

read virtual page from swap dev or exec file;

sleep (event: I/O done);

}

awaken processes (event: page contents valid);

}

set page valid bit;

clear page modify bit, page age;

recalculate process priority;

out: unlock region;

}

For the pages not in the address space and for the pages which are in the address space but do not currently

have a physical page assigned to them of a process, validity bit is not set. The virtual address at which the

fault occurred, it provided to the algorithm. If the disk block descriptor has no reference of the page, the

kernel sends a "segmentation violation" signal to the process.

The page that caused the fault could have the following states:

1. On a swap device and not in memory.

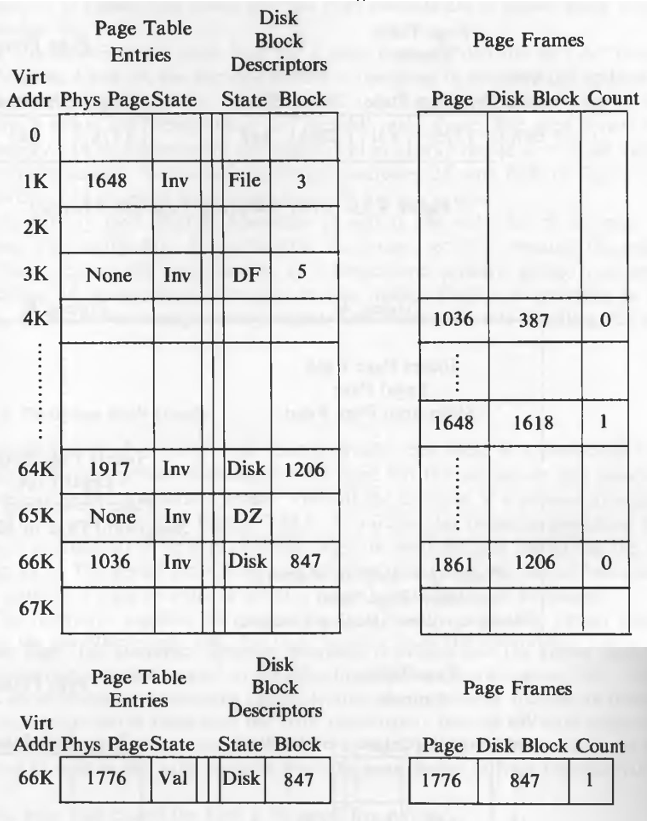
2. On the free page list in memory.

3. In an executable file.

4. Marked "demand zero".

5. Marked "demand fill".

Let us see each case in detail. Consider the following states of data structure:



In case 1, the page is on the swap device, that means it once resided in the main memory. From the disk

block descriptor, the kernel finds the swap device and block number where the page is stored and verifies

that the page is not in the page cache. The kernel updates the page table entry so that it points to the page

about to be read in, places the pfdata table entry on a hash list and reads the page from the swap device.

For example, consider the page table entry for virtual address 66K in the diagram above. If a process incurs

a validity fault when accessing the page, the fault handler examines the disk block descriptor and sees that

the page is contained in block 847 of the swap device (assume there is only one swap device): Hence, the

virtual address is legal. The fault handler then searches the page cache but fails to find an entry fro disk

block 847. Therefore, there is no copy of the virtual page in memory, it must be read from the swap device.

The kernel assigns page 1776 (see the figure above), reads the contents of the virtual page from the swap

device into the new page, and updates the page table entry to refer to page 1776. Finally, it updates the disk

block descriptor to indicate that the page is still swapped and the pfdata table entry for page 1776 to indicate

that block 847 of the swap device contains a duplicate copy of the virtual page.

In case 2, the page is on the free list. It then ignores the block number in the disk block descriptor and

reassigned the page table entry to point to the page just found, increments its page reference count, and

removes the page from the free list. For example, the entry at virtual address 64K, describes that the page is

not in memory, but at 1206 block on the swap device. But when kernel looks if the page is in the cache, it

finds that the page 1861 contains the contents of block 1206 of the swap device. That means that the page is

found in the cache. It resets the page table entry for virtual address 64K to point to page 1861, sets

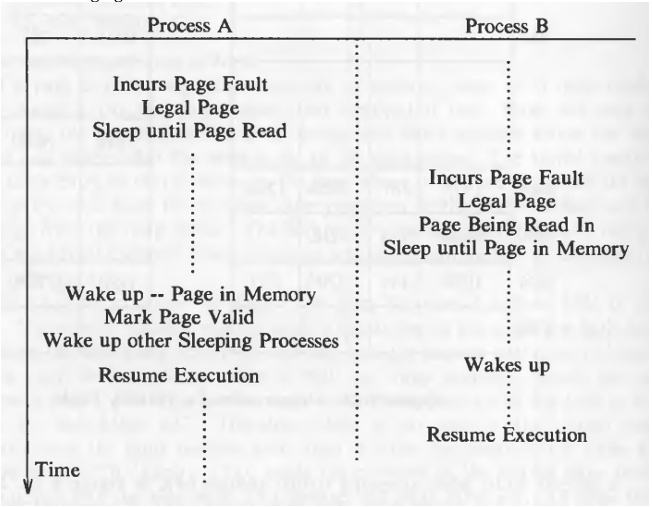
the valid bit, and returns. The disk block number thus associates a page table entry with a pfdata table entry,

explaining why both tables save it.

If a process is trying to read a page while other process just faulted over it and is reading the page in, the

process trying to read the page sleeps because the region is locked by the process which is reading page in.

The following figure shows such condition:



If the page is not on in memory neither in swap space, it has to be read from the executable file (case 3). The

fault handler examines the disk block descriptor, finds the logical block number in the file that contains the

page, and finds the inode associated with the region table entry. It uses the logical block number as an offset

into the array of disk block numbers attached to the inode during exec. Knowing the disk block number, it

reads the page into memory. In the example figure, the virtual address 1K points that the contents of that

page are in logical block 3 in the executable file.

If a process incurs a page fault for a page marked "demand fill" or "demand zero" (cases 4 and 5), the kernel

allocates a free page in memory and updates the appropriate page table entry. For "demand zero", it also

clears the page to zero. Finally, it clears the "demand fill" or "demand zero" flags: The page is now valid in

memory and its contents are not duplicated on a swap device or in a file system. Virtual addresses 3K and

65K in the example figure are like that.

The validity fault handler also sets the valid bit and clears the modify bit. It recalculates the process priority,

because the process may have slept in the fault handler at a kernel-level priority, giving it an unfair

scheduling advantage when returning to user mode. Finally, if returning to user mode, it checks for receipt

of any signals that occurred while handling the page fault.

4. Explain in detail allocation of space on swap device. 286

5. Explain the function of line discipline and clists?

6. Explain the swapping od a process between swap space and main memory? 290

8. Write a short note on: Streams. 358