CHAPTER 9 MEMORY MANAGEMENT POLICIES

Introduction :- If we want a process to run, at least some part of it must be in main memory. Because CPU can not run a process that is entirely in secondary memory(i.e. secondary storage device).

Main memory is crucial because it is the place where program execution takes place. So if you have 8 MB of main memory and you try to run 9 MB process of 1 MB size each, they wont fit in memory.

So operating system(i.e. kernel) has separate special part which deals with effective use of memory. That part is called as Memory Management Subsystem. This part follows some strict set of rules for effective management of memory of processes. These rules together called as memory management policies.

These policies decide which process should reside in memory (either partially or completely) and thus manage that part of process accordingly which is right now not in main memory if process is completely not loaded .

So principally, memory management subsystem monitors the available amount of primary/main/physical memory and if more memory is required, it writes processes to secondary memory(also called as secondary storage device or swap device) to provide more space in main memory. Later on kernel can read these processes from swap device to bring them back to main memory.

Swapping :- When the entire process is transferred to the swap device & then read back entirely from the swap device into the main memory, then this memory management policy is called as swapping. Some exception to the word "ENTIRELY" is there. Means when a process has its text region as "SHARED", then it is kept in main memory and remaining all process is swapped to swap device.

This policy was good for those machines on which "MOX PROCESS SIZE" is quite less E.g. DEC PDP 11, on which max process size is just 64 KB. Thus it is clear that for this policy process size is bounded by amount of available memory.

Swap device **!** ***Virtual memory.***

Paging system

Demand paging:- BSD(release 4.0) was the first to introduce different policy called as demand paging. In this policy not the entire process but memory pages are transfers to and from the swap device recent release of UNIX, such as system V also started supporting this policy.

Here suppose considerable number of pages are already transferred to swap device and kernel now needs them to reload in main memory, then kernel can demand and then load required number of pages only to which process is right now referencing.

The advantage of demand paging is that, it gives greater freedom for mapping of virtual address space of a process into the physical memory. (i.e. memory pages) of a machine. This allows size of a process to be greater than the amount of available physical memory thus allowing more processes to be resided in memory simultaneously.

While advantage of swapping policy is that, it is easier to implement and requires less system overhead. Obviously demand paging requires more system overhead and it is difficult to implement than swapping.

Swapping :- Swapping algorithm is divided into 3 parts...

(a) Managing space on the swap device.
(b) Swapping processes out of main memory and
(c) Swapping process into the main memory.

(a) <u>Managing space on the swap device</u> :- The swap device (or secondary storage device or secondary memory) is actually a part of disk which is pre-planned kept aside. So it is a configurable part of disk and in terms of kernel it is a block device. Means its unit is also "A BLOCK" as like file system allocated hard disk.

But differences between "A FILE SYSTEM BLOCK" and "A SWAP DEVICE BLOCK" are very important ....(5)

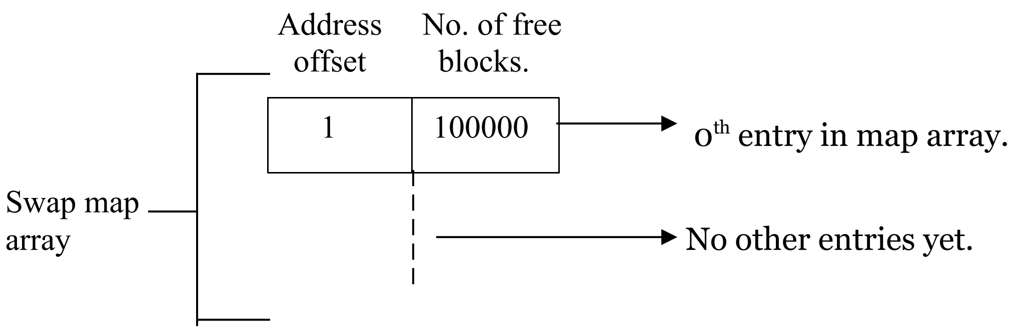
- For a file kernel allocates "ONE BLOCK" at a time and when it is filled completely, then and then only it allocates another block. But for swap device kernel allocates space in groups of continuous blocks.
- Space allocated for file is static and hence exist long time. But the space allocated for swap device is temporary (i.e. transitory), dynamic and hence charges continuously due to incoming and outgoing processes to and from it respectively.
- As allocation for file is static, the allocation method is such that there is less fragmentation as possible. Thus unallocated, discontinuous place is less. Whereas the allocation for swap device is transitory and "FAST CHANGING", hence much of fragmentation occurs here, because "LEARNING PROCESSES" free the swap space at irregular sequence and at irregular intervals depending upon the process scheduling algorithm.
- But on swap device fragmentation is ignored deliberately because swap device is concerned with a process and not with a file. So time is crucial factor for performance by processes. Thus I/O from swap device must be very fast than the I./O on file system. That is why to make I/O faster, the "GROUP OF BLOCKS" i.e. multi block allocation is used. So that one multi block operation can do I/O faster than reading or writing a single block one at a time for several times. Thus now it is clear that kernel allocates continuous space on swap device without thinking about the fragmentation.
- As allocation scheme for file system and for swap device, differs significantly, the data structures for swap device are also quite different. E. g. :- For file system kernel maintains free space as linked list of free blocks which is accessible through the super block. But for swap device, the information about free space is kept in main memory (not in super block) as an array. (recall, "CONTINUOUS ALLOCATION IS DONE FOR SWAP DEVICE").

<u>Map</u> :- As mentioned in last paragraph, the free space on swap device is kept as an array in main memory (i.e. so called in-core table).

This array is called as map.

Each entry of this array contains starting address of an allocate able block and number of available blocks. Here kernel treats address as a block offset from the beginning of swap device.

So initially (i.e. When whole swap device is empty, means yet no any process is swapped here) map will contain only one entry with address offset 1 and corresponding free blocks in empty swap device. If we assume that there 10,000 free blocks in swap device, the figure of this single map entry will be...



As kernel goes on allocating and de allocating these 10000 free blocks, it updates the map array in such a way that, it always contain accurate information about number of remaining free blocks.

The algorithm for allocating space from map is called as malloc(), which is as follows...

```
algorithm : malloc() /* algorithm to allocate map space */
input  : (1) Map address
         (2) requested number of free blocks
{
         for(every map entry)
         {
                 if(current map entry can fit requested block)
                 {
                   if(requested blocks == number of blocks in current map entry)
                           delete this entry from map array;
                   else
                           adjust start address of map entry;
                   return(original address of map entry);
                 }
         }
         return(0);
}
output : if successful, returns address
       otherwise 0 (i.e. NULL)
```

- Kernel searches the swap map array each entry at a time, un till it finds the entry which has enough number of free blocks to fulfill the requested number of free blocks.
- If number of free blocks in an entry is exactly equal to the number of requested free blocks, kernel removes this whole entry from the map array and returns its address to the caller so that now caller can use his requested number of free blocks starting from the given address (i.e. address returned by malloc());
- As one whole entry gets removed from the map array, kernel compresses the map array. Means now on words the number of elements in array will be less than 1 from the previous number of elements.
- But if number of requested free blocks does not match with any entry, it adjust the address and number of remaining free blocks (assuming that requested blocks are removed from total available blocks) and returns address same as for previous condition.
- If kernel really can not give required number of blocks, it returns 0 i.e. NULL. Obviously this will happen when call requested for such number of free blocks which is more than that of the available number of free blocks on swap device.

Example :-

1	10000

$\longrightarrow$ Original 1 entry before any allocation

Now suppose a request of 100 free blocks is done.

101	9900

$\longrightarrow$ Here as 100 are removed from total 10000 and from address 1, now 9900 are remaining from address 101.

1 + 100   10000 – 100

Mathematically :-

So for address – add requested number to present number.
For blocks – subtract requested number from present number.
Now suppose a request of 50 free blocks is done.

151	9850

$\longrightarrow$ Here, as 50 blocks are removed from total 9900 and from address 101, now 9850 are remaining from address 151.

101 + 50 9900 – 50

Now suppose more 100 free blocks are requested.

251	9750

$\longrightarrow$ Here, as more 100 blocks are removed from total 9850 and from address 151, now 9750 are remaining from address 251.

Above example shows that, at last i.e. after 3 requests, the swap device had allocated total of address 1 to address 250 space of swap device. In terms of blocks, it allocated 250 free blocks. (100 + 50 + 100). So finally now 9750 blocks are available starting from address offset 251.

Up till we saw about how malloc() can be used to allocate space from the swap device. So allocation of space from swap device, decreases number of available resources on the swap device.

When freeing resources kernel finds proper position in the map by addresses.

Now what will happen when allocated resource are freed ? Obviously they will add to the free resource and number of free resource on swap device will increase. But the scenario is slightly complicated than the allocation.

There 3 possible cases of freeing allocated swap device blocks…

a) **If freed number of blocks are completely accommodable to fill a hole in the map :-** In such a way that the starting address of "FREED GROUP OF BLOCKS" is continuous with the map entries which is above the hole and which is bellow the hole , then kernel plugs "FREED GROUP OF BLOCKS" in the hole and joins all the 3 entries i.e. upper entry + entry plugged into hole + lower entry, to form on single entry.

Means in case (a) suppose there are 2 map entries and a hole, when one freed is plugged in, number of entries does not became 3, but become 1 number of map entries gets decreased due to combination of all.

b) **If the freed number of blocks partially fill a hole in the map in such a way that the address of "FREED GROUP OF BLOCKS" is continuous either with the address of upper entry or with the address of lower entry but not with the both (as in case (a)) :-** Here kernel plugs freed blocks in such a way that the continuous upper or lower entry is combined with the "FREED GROUP". Means number of map entries will remain same even if one is added . i.e. Suppose there are 2 map entries and the new freed one is continuous with the lower entry, then freed group is combined with the lower entry. So now in map there are still 2 entries one upper + one combined of freed and lower.

c) **If the freed number of blocks partially fill a hole in the map in such a way that the address of "FREED GROUP OF BLOCKS" is neither continuous with the address of upper entry nor with the address of lower entry :-** Here kernel creates a new entry in map and inserts it in proper position. So obviously in this case number of map entries is going to increase.

To understand above 3 cases we will proceed with the same previous

Example :-

251	9750

The original entry left by previous example.

Now suppose 50 blocks are freed starting at address 101.

101	50
251	9750

Number of entries increased as one is added.

Here, 50 blocks are freed at address 101. Address 101 is not continuous with address 251, because 101 + 50 gives 151 which is not 250 to be continuous with 251. Thus new entry is needed. This entry is placed above previous one because 101 is less than 251.

Now suppose that 100 blocks are freed starting at address –1.

1	150
251	9750

Number of entries remain same as the freed entry and one existing entry Is combined into a single entry.

Here 100 blocks are freed, starting at address 1. Now if we add 100 to 1 it gives 101st address, which is continuous with lower address 101 (see above case's figure). So this entry i.e. 1 – 100 and the lower entry 101 – 50 can be combined together to form a combined together to form a combined entry like 1 – 150. Now as 1 is lesser than 251, this entry is added above the entry of 251 – 9750.

<u>Assumptions for above example</u> :- Words upper and lower are used assuming the array longitudinally for the sake of simplicity. Word "LESSER" is used because we talk about addresses as offset so using this word makes understanding simpler.

The example of malloc() seen on Page 233 and page 234 assumes that this is the beginning of use of swap device and hence there is only one entry.

But what will happen if kernel takes(or allocates) block from other entry, if map has multiple entries ?

Consider our same example. Right now there are 2 map entries ....

1	150
251	9750

The original entry left by previous example.

Now suppose kernel requested 200 bocks.

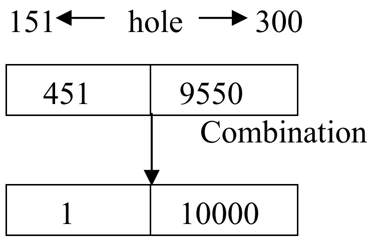
1	150
451	9550

Here, the request of 200 blocks can not be satisfied by first entry 1 – 150, as it has only 150 blocks. So kernel allocates 200 blocks from 2nd entry 251 – 9750 which has 9750 blocks. After allocating 200 blocks, the address will become 451 (251 + 200) and blocks will reduce to 9550 (9750 – 200).

Now suppose kernel frees 300 blocks starting at address 151.

1	150

Here we know that, the hole's address begins at 151 (1 + 150) and out of 10000, 9700 are free (150 + 9550), so



300 are still allocated. When kernel wants to free 300, it realizes that request perfectly fits the hole and thus combines all to give one single entry.

One single entry

Older UNIX versions used only one swap device but latest UNIX versions, i.e. system V allows multiple swap device too.

If there are multiple swap devices, kernel use them in round robin scheme provided the usable swap device has enough space.

Administrator can create and remove swap devices dynamically. If a swap device is removed, kernel first stops sending data to it. But if it already has some data, it first empties it and then remove it. So there may be a gap between issuing command of swap device's removal and actual swap device removal by kernel.

(b) <u>Swapping Process Out Of Main Memory</u> :- Kernel swaps a process out of main memory when it needs more space in memory due to….

(a) The fork() system call is mode to create a child process.
(b) The brk() system call increases the size of a process.
(c) A process becomes larger by natural growth of its stack.
(d) Kernel wants to free space in memory for reloading of those processes which it had previously swapped out and should now swap in.

Algorithmically the process of swapping out of a process is as follows …….

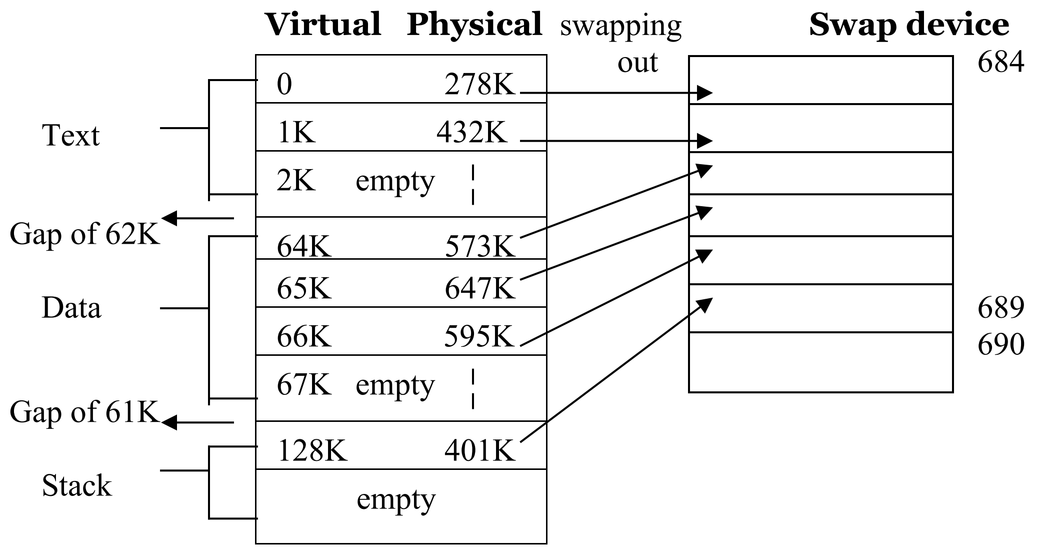
- ✓ When kernel decides that a process is now eligible to swap out, it first decrements the reference count of each of its 3 regions and then swaps the region out to swap device only if its region references count (Obviously after decrementing) drops to 0.
- ✓ As per the need of this swappable process, kernel allocates enough space on swap device.
- ✓ As yet the process is not swapped (means it is swappable but not yet swapped), means still it is in memory, kernel locks it, to prevent "SWAPPER PROCESS" to swap it out because some tasks are yet left to be done before its swapping.
- ✓ Kernel saves the swap address (i.e. the address on swap device where regions are going to be put after swapping) of each region in region table entry of this process.
- ✓ The kernel starts swapping of as much data as possible from process's (swappable) user area to the swap device in each I/O operation. **Note that** this is a direct I/O, means no help from buffer cache is taken for this I/O because time element is crucial.
- ✓ If hardware constraints does not allow transfer of multiple pages in single operation, then kernel must do it iteratively (i.e. in loop) by transferring one page of memory at a time. Thus rate of transfer depends upon hardware capabilities such as capabilities of disk controller (recall that swap device is a part of disk). E.g. :- As pages in page table may be discontinuous (rather they are), then kernel first must gather those discontinuous page addresses concerned with the swappable data.

  It gathers "PAGE ADDRESSES" not actual page contents, because system is already storing from memory.

  Then disk driver uses this "COLLECTION OF PAGES" to do disk I/O. Also **note that**, swapper process waits for such individual I/O of data before starting next I/O of swapping of next data.
- ✓ The word "ENTIRE" used in the introduction of swapping on Page 230, does not mean "ENTIRE VIRTUAL ADDRESS SPACE OF THE PROCESS". Because the total virtual address space of a process contains 2 types of processes i.e. assigned and unassigned. Out of

these two, kernel copies only assigned virtual addresses to the allocated space on swap device, ignoring the unassigned addresses. Here a question may arise, how kernel could know about the addresses when it will swap the process back into the memory ? This is because kernel keeps the map of virtual addresses of process before swapping it out. So when kernel swaps the process back into the memory, it can re-assign the process to the correct virtual addresses. Thus process also can read the data back to correct address locations eliminating the need for any special data buffer memory.

Layout of virtual addresses



Note :- For simplicity the virtual address space in this figure is shown as linear array of page table entries. Actually each region has own page table.

This figure is an example of mapping of “VIRTUAL ADDRESS SPACE” of a process in memory to the swap device.

In this figure “TEXT REGION” begins at virtual 0K (physical 278K) to 2K the entry of 2K is empty. Means text region has 2 assigned entries and one un-assigned entry.

“DATA REGION” begins at 64K (physical 573K) to 67K. Out of its 4 entries 64K, 65K, 66K are assigned while entry at 67K is empty.

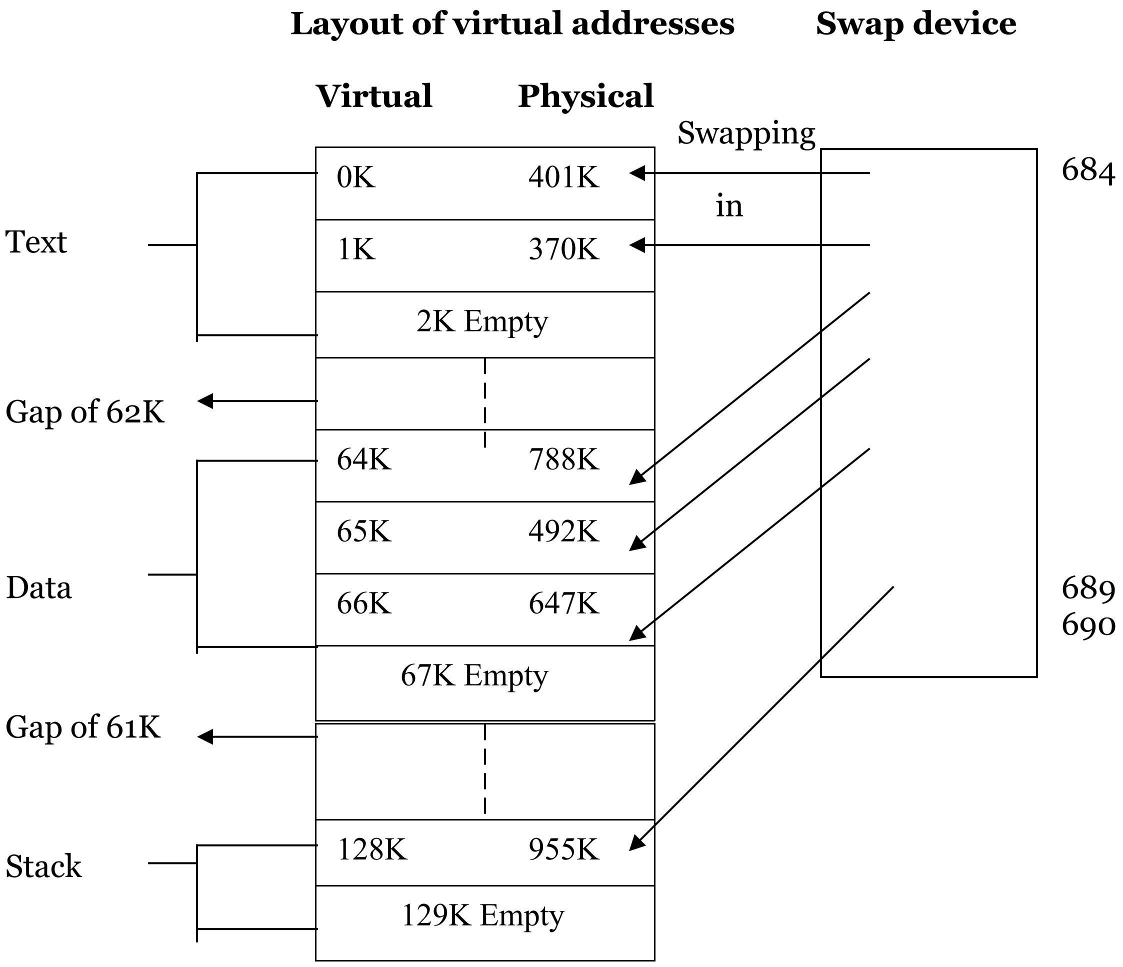
“STACK REGION” begins at 128K (physical 401K) to 129K, where entry at 128K is assigned and entry at 129K is empty.

*** Some important things to note :-**

- Virtual addresses for a region are continuous (as they must be). Means text region is from 0 to 2K, data region is from 64K to 67K.
- But their physical addresses i.e. pages are not continuous. Means 0K corresponds to 278K physical page. So we may expect that 1K should correspond to 279K physical page. But actually it is 432K physical page. So non-continuous.
- Only assigned entry of region has physical page. The non assigned entry i.e. “EMPTY”, does not have physical page.
- Entries in a region are continuous. This does not mean that 3 regions should be continuous with each other exactly. Means they are continuous but may keep gap between them to allow a region to grow whenever required. E.g. :- Text region ends at 2K while Data region does not begin from 3K, it begins from 64K, leaving a gap of 62K in between these 2 regions. Same can be observed for end of Data region and beginning of stack region. Now we will see about mapping. When kernel swaps the process out, it memorizes whole “VIRTUAL ADDRESS MAP”

so that when process swaps in, it could reassign correctly. While swapping kernel swaps only the entries (or physical pages) for virtual addresses 0K, 1K, 64K, 65K, 66K, and 128K only. Because they are the only assigned virtual addresses to actual physical pages. It does not copy "EMPTY" entries neither the gaps between regions (i.e. gap of 62K between Text And Data regions or gap of 61K between Data and Stack regions). So obviously entries in "SWAP DEVICE" become strictly continuous (rather it is the most important requirement of swap device) from location 684 to location 689 of 6 entries.

Now when process gets swapped back into the memory, kernel has memory of all "VIRTUAL ADDRESS MAP" of the swapped process. Not only that, but it also knows that there was a gap of 62K between Text and Data region and a gap of 61K between Data and Stack region.



Look at the differences in figure of "SWAPPING OUT" (on Page 238)and "SWAPPING IN". The physical pages are different. Means when "SWAPPING OUT" physical page was 278K for 0K virtual address which becomes 401K when "SWAPPING IN".

This is because, process gets swapped out due to some memory requirement for other process. When time comes, process gets swapped back in, out at this time previous pages (at the time of swapping out) may get allocated some other process.

Thus "SWAPPING IN" may get different physical pages but **Note that** as virtual map is same, there is no problem by assigning available different pages. So important is "VIRTUAL SPACE" assigned to the process and not the physical pages as physical pages can be re-assigned.

Note :- Theoretically any part of the process including its "U AREA", kernel stack is swappable. But "LOCKED REGIONS" most commonly the "TEXT REGION" if shared, not swapped out.

Also kernel practically do not swap "U AREA" if "U AREA" has the "VIRTUAL ADDRESS MAP" which is required when process swaps back into memory. This map is of utmost important because it actually has "ADDRESS TRANSLATION TABLE" which tells the kernel about mapping of process's virtual addresses to physical pages.

Fork swap :- The algorithm of fork() which we had seen (on Page 166 - 171)assumes that there is enough memory to create a child. I.e. there is enough memory for child's context.

But if, rarely, but sometimes, kernel found that there is no enough memory for child, then kernel swaps the child out to the swap device. But this time kernel does not free its occupied memory (means though swapped out, child's memory is kept intact and thus not given to any other process to occupy).

When swapping of child completes, the child exists but not in memory but on swap device. The parent during fork(), keeps the child on swap device as "READY TO RUN" and then completing fork() parent returns to user mode.

As child on swap device is in "READY TO RUN" state, and when swapper process swaps it back in memory, kernel schedules it and child starts running in memory. Now child completes its part of fork() and returns to user mode.

Expansion swap :- We know that a process can call brk() system call to grow its data region or a process's stack region can automatically grow. For these two purposes, where process's size gets increased, more memory is required.

Means in such cases process need more memory than the current available memory for it. If memory is available, it is given. But if some memory constraints are there, the kernel does "EXPANSION SWAP" for this process.

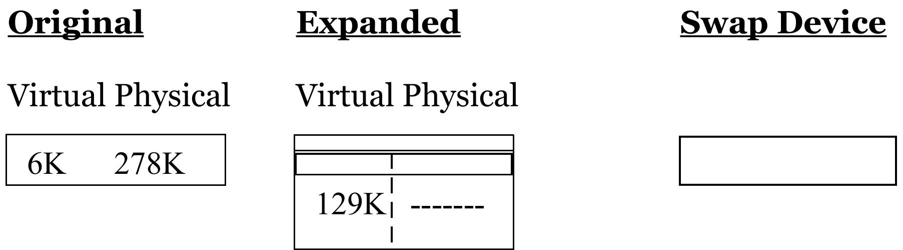
During expansion swap, as usual kernel allocates enough space on swap device for this process including the newly required extra memory's space.

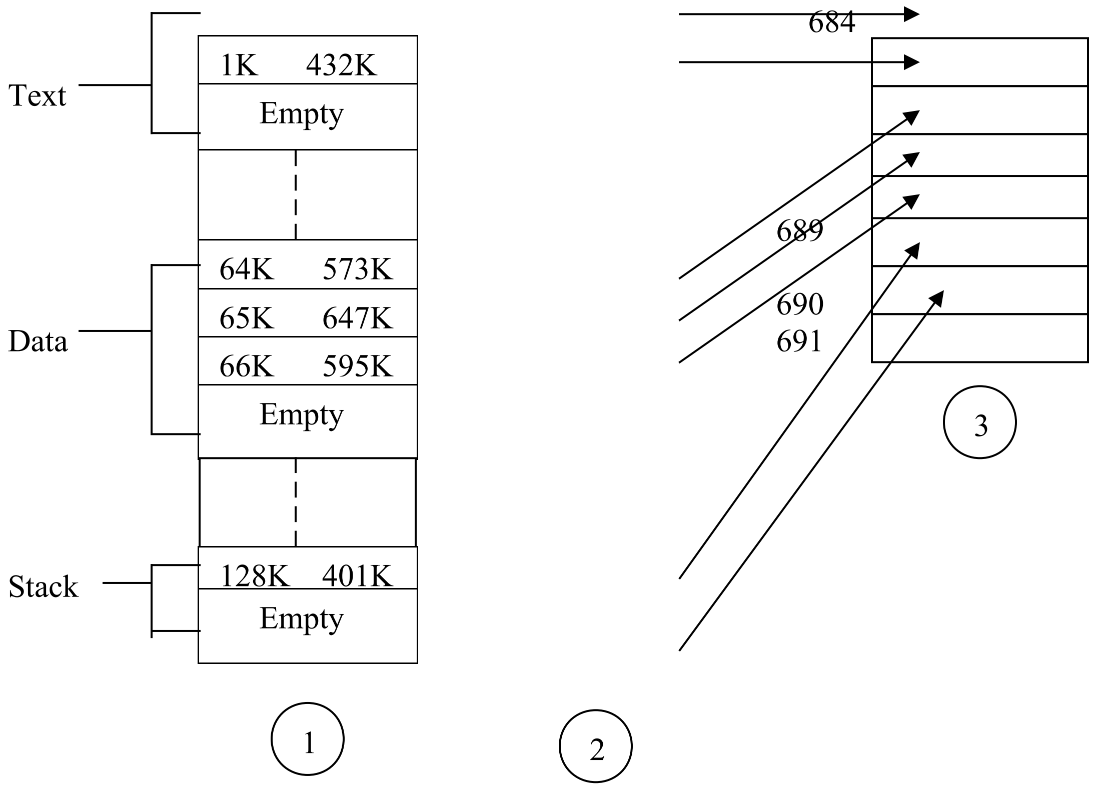
Then it gives "VIRTUAL ADDRESS" to this new space in respective region (i.e. according to region which is growing means Data or Stack) but does not assign any physical page to this new space.

Finally it swaps out the process as usual and "ZERO OUT" the contents of "NEW SPACE" on swap device.

Now when the kernel swaps the process back into the memory, then it actually assigns physical page to the new space by using "ADDRESS TRANSLATION" method and process now starts executing (as it is now in memory) with its requested extra space.

See below figures. In figure (1), original layout of "VIRTUAL ADDRESS SPACE" of process is shown. Now suppose process wants to grow its stack region by 1 page of memory (actually stack grows automatically). So in the figure (2) i.e. of "EXPANDED LAYOUT", a new page with virtual address assigned to 129K is added. (here it is assumed that stack grows towards higher address). But no physical page is assigned. (on swap device, this entry goes at 689 location, which is the 7th entry).





(c) <u>Swapping process into the memory</u> :- "PROCESS 0" is the only process which swaps processes back into the memory from swap device. As we know after forking process 1, process 0 executes algorithm of "SWAPPER" to become "SWAPPER PROCESS".

Not only the function of "SWAPPING IN" but also the function of "SWAPPING OUT" of processes is done by this "SWAPPER PROCESS".

The "SWAPPER PROCESS" sleeps if it there is no work of swapping to do, or it sleeps when it can not do the work. E.g. If it is awakened to swap a process "IN", but if there are no "SWAPPED OUT" processes on swap device or if it is awakened to swap a process "OUT", but there is no such eligible process in memory which should be swapped out. In both these circumstances swapper process sleeps.

This is because, sleep of "SWAPPER PROCESS" is not wake able by events (un like other process which wake up from sleep on some event completion) but kernel wakes it up "PERIODICALLY" irrespective of whether there is really a "SWAP IN" or "SWAP OUT" process exists.

With reference to kernel, swapper process is just like other processes, hence scheduling swapper process is similar to the scheduling other processes.

But there are 3 major differences by which kernel can treat it differently than that of other processes.

(1) Swapper process always run in kernel mode only.
(2) Swapper process has higher priority than other processes.
(3) Swapper process does not make any system calls. Instead it uses kernel's internal algorithms to do swapping. So obviously swapper process is of "KERNEL PROCESS" type.

When swapper process is worked up to swap a process "IN", it examines all the processes in swap device (which are swapped out previously). Then it looks out of "SWAPPED OUT" processes which in the state of "READY TO RUN BUT SWAPPED OUT", then it selects one of them which fulfills the criteria of "BEING SWAPPED OUT FOR LONGEST TIME" and then swaps that process "IN", if enough free memory is available.

The process of "SWAPPING IN" is the reversal of the process of "SWAPPING OUT". Means it allocates physical memory, reads the process from swap device into the allocated physical memory and finally frees the relevant portion of swap device.

It repeats the same above task again and again until….

A. There remains no “READY TO RUN BUT SWAPPED OUT” processes on swap device. In this case swapper process goes to sleep until kernel swap out a process on swap device which is scheduled to “READY TO RUN BUT SWAPPED OUT” state.

B. The swapper process finds a process in “READY TO RUN BUT SWAPPED OUT” state on swap device, but it can not do “SWAP IN” of this process because there is no enough memory. In this case swapper process first tries to swap another process out and if it does this successfully, then restarts swapping algorithm to search for a new process for “SWAP IN”.

For “SWAPPING OUT”, if there is such a situation, where the swapper process must swap a process “OUT”, it examines every process in memory, the process for which it is looking, must not be zombie, because zombie process is never swapped (neither “OUT” nor “IN”), because zombie process never occupy any physical memory. The process for which it is looking, must not be locked (the process which is doing some “REGION CONCERN OPERATION” is always locked).

```
algorithm : swapper() /* swap processes “in” & “out” */
input : nothing
{
    loop 1:
          for(all “swapped out” processes which are in “ready to run” state)
               pick up that process which is swapped out longest;
          if(no such process)
 {
                  sleep(event : wait for some process to enter in swap device “as swapped out” &
                    then must swap in);
                  goto loop 1;
          }
          if(enough room in memory for a process to “swap in”)
          {
                  swap this process “in”;
                  goto loop 1;
          }
    loop 2 :  /* as per revised algorithm */ previously it was loop 1:
    for (all process loaded in memory, but not zombie, and not locked in memory)
     {
          if(there is sleeping processes in memory)
                  choose a process if “priority + residence time” is numerically highest;
          else /* means not sleeping process */
                  choose a process such that, if “residence time + nice” is numerically highest;
     }
     if(chosen process not sleeping or “residence time + nice” is not satisfied)
          sleep (event : do swap in);
     else
          swap out the process;
      goto loop 2 ; /* as per revised algorithm*/
}                                      previously it was loop 1:
 output : nothing
```

While choosing a process “TO SWAP OUT”, kernel chooses such a process which is not liable to get scheduled quickly. That is why it prefers “SLEEPING PROCESS” to swap out rather than some “READY TO RUN” process. Because “READY TO RUN” process has greater chance of being scheduled quickly rather than sleeping process. So if more than one process are sleeping in memory, the choice of which to choose is made on a function of “SUMMATION OF PROCESS’S PRIORITY AND THE TIME FOR WHICH IT IS IN MEMORY”. The numerically highest answered process is swapped out.

Now if there are no sleeping processes in memory, and if “SWAPPING OUT” is must, then swapper chooses such a process whose “SUMMATION OF MEMORY RESIDENT TIME AND NICE VALUE” is numerically highest.

As a hard coded rule, the process (not sleeping) must be resident in memory for ≥ 2 seconds to become eligible for “SWAPPING OUT”. Similarly such a process can “SWAP IN”, which is previously swapped out for ≥ 2 seconds.

Now if swapper can not found...

- Any process to swap out.
- Any process to swap in.
- Any process in memory is yet not crossed 2 seconds mark to “SWAP OUT”.
- Any process in swap device is yet not crossed 2 seconds mark to “SWAP IN”.

Then swapper process sleeps on event that it wants to swap a process in but does not have enough memory.

In this “SLEEPING STATE OF SWAPPER PROCESS”, clock awakens it continuously once a second.

This does not mean that only clock can awake it. Kernel also can awake it if any other process goes to sleep, thinking that this slept process might be eligible for “SWAP OUT”, than the process under consideration currently.

Note that :-

! Whenever swapper swaps a process out or

! Whenever swapper sleeps because it can not swap any process out, ...then swapper resumes its execution from the beginning of the “SWAPPER ALGORITHM” attempting to swap eligible process “IN”.

Example **Processes**

Time In Seconds	A	B	C	D	E
Beginning state 0	0 runs	0	Swapped out (0)	Swapped out (0)	Swapped out (0)
1	1	1 runs	1	1	1
2	2 Swapped out (0)	2 Swapped out (0)	2 Swapped in (0) run	2 Swapped in (0)	2
3	1	1	1	1 runs	3
4	2 Swapped in (0)	2	2 Swapped out (0)	2 Swapped out (0)	4 Swapped in (0)
5	1 runs	3	1	1	1
6	2 Swapped out (0)	4 Swapped in (0)	2 Swapped in (0)	2	2 Swapped out (0)

Runs

Sequence of swapping operations.

<u>Assumptions</u> :- For 5 processes A, b, C, D, E……
(1) All processes do not make in system calls.
(2) As there are no system calls, context switch is assumed to appear after every 1 second by clock interrupt.
(3) Swapper process is running at highest priority.
(4) As swapper process has highest priority, it will run exactly after every 1 second if it has work to do.
(5) Processes are of same size.
(6) System can contain maximum 2 process in memory at a time.
(7) Initially processes A and B are in memory, and other processes i.e. C, D, and E are swapped out.

- ❑ For first 2 seconds swapper can not swap any process (neither "IN" nor "OUT"), because neither any process is in swap device for more than 2 seconds nor any process is in memory for more than 2 seconds.
- ❑ At the first "2 SECOND" mark, swapper swaps out both A and B as they both touch the 2 seconds mark. Obviously swapper will swap processes C and D into the memory.
- ❑ It tries to swap E "IN", but fails because memory is assumed to contain only 2 processes at a time and hence there is no enough space in memory for E.
- ❑ At the "3 SECONDS" mark, again process E is eligible to "SWAP IN", but as processes C and D yet not touched or crossed their "2 SECOND" limit, hence they can not be swapped out and thus there is no enough memory for E.
- ❑ At the "4 SECONDS" mark, the swapper swaps out processes C and D as both touched their "2 SECONDS" limit and also swaps processes E and A in memory as there is now room for two processes to arrive in memory.

  Up till now we saw that kernel chooses such a process to "SWAP IN", which spends ≥ 2 seconds of time on swap device after swapped out. So this is a "TIME-BASED" choice. Another criterion was suggested that, a process can be "SWAPPED IN" if it is "READY TO RUN BUT SWAPPED OUT" state and having highest priority. This is because such process always has greater chance of being scheduled quickly. This policy results in better "SYSTEM PERFORMANCE" under heavily loaded systems.

<u>Algorithm for "SWAPPING OUT" in "SWAPPER ALGORITHM" has more flaws</u> :-

1) Algorithm states that swapper selects a process to get "SWAP OUT" on basis of
   - ! Priority.
   - ! Memory resident time.
   - ! Nice value.

   Here a major issue arises that, actually "SWAPPING OUT" is done to make some memory free for incoming processes. But criterias above stated may swap such a process which can not fulfill the "NEED OF MEMORY" of incoming process. E.g. :- There is an incoming process which requires 1MB of memory. Then it makes no sense to "SWAP OUT" a process of 2K size but fulfill the above 3 criteria's.

   An alternative to above flaw was suggested that, instead of "SWAPPING OUT" one process of 2KB size, "SWAP OUT" such a group of processes which fulfill above criteria and as well as they all will free more than or equal to 1Mb memory together for the incoming process of 1 Mb size. This alternative can increase the system performance much better. *This change is done in our revised algorithm*.
2) If swapper process sleeps because it can not find enough memory to "SWAP IN" a process which it had already chosen as "ELIGIBLE TO SWAP IN". But algorithm says that after waking up the swapper must again enter into the loop to search for eligible "SWAP IN" process. Here it may happen that, mean while other "SWAPPED OUT" processes may be awaked and more eligible to "SWAP IN" than the previously chosen one.

   Hence our revised algorithm works well here for this flaw. Because in revised algorithm we try to "SWAP OUT" as many smaller processes to make room for the incoming bigger process.

So there is less chance for the swapper to go to sleep due to "NO ENOUGH ROOM IN MEMORY" for "SWAP IN" process.

3) While "SWAPPING OUT", kernel first searches slept processes, if there is no sleeping process, it searches other processes which are not sleeping but following the 3 criteria (i.e. priority, memory resident time and nice value).

   Now suppose among these "NO-SLEEPING PROCESSES", if by mistake swapper chooses such a "READY TO RUN" process which is not executed yet after its "SWAP OUT". Obviously this will be "INJUSTICE" for this process.

	A	B	C	D	E
0	0 runs	0	Swap out (0)	Nice 25 Swap out (0)	Swap out (0)
1	1	1 runs	1	1	1
2	2 Swap out (0)	2 Swap out (0)	2 Swap in (0) runs	2 Swap in (0)	2
3	1	1	1	$\boxed{\text{1 Swap out (0)}}$ **Injustice**	3 swap in (0) runs
4	2 Swap in (0) runs	2	2 Swap out (0)	1	1
5	1	3 Swap in (0) runs	1	2	2 Swap out (0)
6	2 Swap out (0)	1	2	3 Swap in (0) runs	1

In above diagram kernel swaps in process D at the "2 SECOND" mark . then schedules processes C. So memory is filled due to over assumption that memory can have at most 2 processes at a time in memory.

Now at "3 SECOND" mark, process E becomes eligible to swap in, actually swapper should sleep because there are already 2 processes in memory and thus there is no space.
But as process "D" has its nice value 25, it is the lowest priority process than E, hence swapper again out process D for "SWAP IN" of E.

Here you can see that after D's swap out, it gets "SWAP IN" at "2 SECOND" mark, but before completing its 2 seconds in memory it is unjustifiably swapped out again. This is called as thrashing due to swapping.

4) If swapper tries to "SWAP OUT" a process but can not find space in swap device, then a deadlock may occurs, in following (4) cases occur simultaneously.

(a) **All processes in memory are sleeping :-** Here swapper has work to do but swap device is full.
(b) **All "READY TO RUN" processes are swapped out :-** Which will be scheduled?
(c) There is no room on swap device for new process to swap out.
(d) There is no enough memory for incoming processes.

Above, rare but possible dead lock is still there but fixing of above and the last three flaws, is quite ignored because of new scheme of memory management i.e. Demand Paging.

Demand Paging :- If machine's hardware supports 2 features…

- Memory based on pages.
- CPU has restart able instructions :- Means if CPU executes "PART OF AN INSTRUCTION" (i.e. not full instruction) and a page fault occurs, the CPU must restart the whole instruction after handling the fault. This capability of going back to the beginning of instruction is called as "RESTART ABLE INSTRUCTION".

  …..then kernel do memory management by the policy of demand paging, where "NOT THE ENTIRE PROCESS", but only certain number of pages are swapped between main memory and swap device.

  Due to demand paging policy the constraint of process size is absolutely vanished which was there due to the limited amount of actual physical memory. So now on a machine of 1 to 2MB RAM a process of 4 to 5MB size also can run smoothly.

  But still kernel gives limit to the process size but this time not due to the limit of physical memory but due to the limit of amount of virtual memory the machine can address.

  Now if a process has its size greater than the available physical memory, then kernel must load such a process partially in memory and remaining portion of process is kept swapped out on swap device.

  Up till now we know that process executes instructions written in its "TEXT REGION" and manipulates relevant data in its "DATA REGION". But this does not mean that it uses Text region and Data region entirely. As instructions are executed one by one and as Data is manipulate according to the instruction, in reality small portion of Text region and small portion of Data region is used. This is called as "PRINCIPLE OF LOCALITY".

  Denning, a UNIX research fellow, defines "WORKING SET OF A PROCESS" based on this principle of locality. According to him "WORKING SET OF A PROCESS" is the set of memory pages that the process has referenced in its last n memory references. Here the number "n" is called as "WINDOW" of the working set.

  Due to principle of locality, working set of a process is always a fraction of entire process. Thus more processes may fit simultaneously into the main memory than in swapping system. This increases system performance potentially by reducing so called "SWAPPING TRAFFIC".

  So when, if process accesses such a memory page which is not part of its working set, the system will issue a "PAGE FAULT" and during handling of this page fault, the kernel updates the working set and reads the page (for which the fault had occurred) from the swap device if fills necessary. This page fault is called as "VALIDITY PAGE FAULT". When page fault of such type occurs the kernel suspends the execution of process until it reads that respective page from swap device and makes the page accessible to the process. When page becomes accessible, it gets loaded into memory and kernel restarts the instruction from the beginning for which fault had occurred. Thus paging requires to do memory management in 2 parts
- Swap out "RARELY USED PAGES" to swap device.
- Handle the page fault.

Data Structures Used For Demand Paging :- There are 4 Data structures which are used for low level handling of demand paging algorithms.
(1) Page table entries.
(2) Disk block descriptor table.
(3) Page frame data table (also called as pfdata).
(4) Swap-use-table. (all these tables are arrays).

(1) Page Table Entry :- Considering the "PAGE TABLE ENTRY", we again see the contents of it but now in some more detail than seen previously. Region contains page tables.
Each entry of a page table has 3 parts.

Single “PAGE TABLE ENTRY” :-

Physical address	Protection bits	Age	Cp/Wrt	mod	Ref	Validity

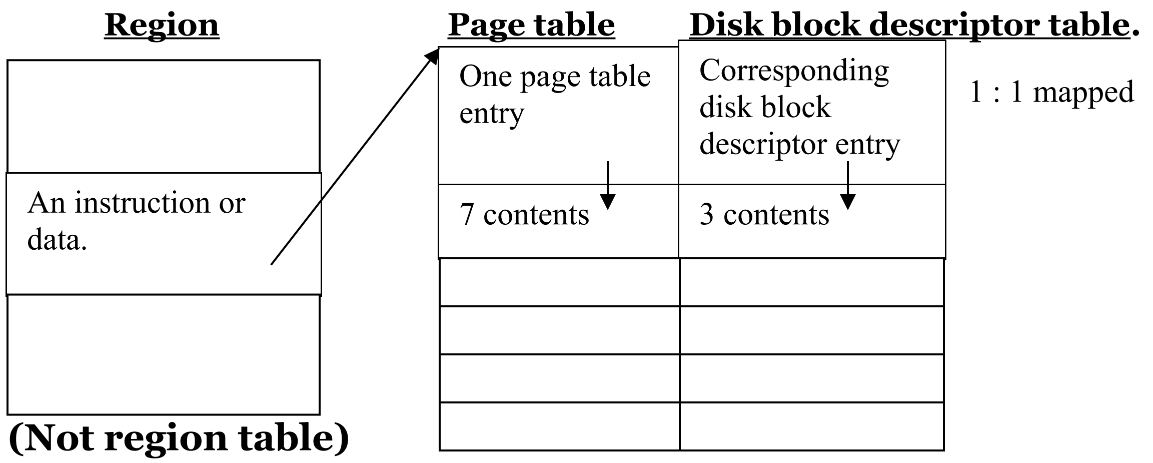
Demand paging concerned bits.

A. Physical address of the page.
B. Protection bits :- Whether process can read, write or execute from this page.
C. Demand paging concerned bits :- There are 5 such demand paging concerned bits....
    1. Valid bit :- Kernel turns this bit “ON” when the contents of this page are legal. But this does not mean that when this bit is “OFF” process can not reference this page. (i.e. not illegal).
    2. Reference bit :- This indicates how many processes reference this page. Means if this is 0, no process has referenced this page yet.
    3. Modify bit :- This indicates whether the process which referencing this page, recently modified the contents of this page or not.
    4. “COPY ON WRITE” bit :- This bit is used in fork() system call on those systems where demand paging possible. This bit specifies that, when it is “ON” kernel must create a new copy of this page when process modifies contents of this page. (also modify bit will get set).
       “COPY ON WRITE”, really means when some process tries to write on this page, make copy of this page and then write on new copy. Don’t touch original.
    5. Age bit :- This indicates how long time this page is member of “WORKING SET” of a process.

Kernel allocates memory for this data structure (i.e. page table entry) dynamically as per the requirement.

(2) <u>Disk Block Descriptor Table Entry</u> :- As like page table made up of page table entries, there is a disk block descriptor table corresponding to each page table.

In other words for one page table entry, there is a corresponding (1 to 1) disk block descriptor table entry.



(Not region table)

Each disk block descriptor table entry contains disk copy of the virtual page. This table is concerned with “SWAP DEVICE”.

Above figure also tells us one important thing. If the region is "SHARED", by two processes, both will reference same "PAGE TABLE ENTRIES" and their corresponding "DISK BLOCK DESCRIPTOR TABLE ENTRIES".

We said that "DISK BLOCK DESCRIPTOR" is mainly concerned with "SWAP DEVICE", because as "SWAP DEVICE" is part of hard disk which is a "DISK BLOCK" device.

But this does not mean that every process has its page only on swap device. Because as process is either running or sleeping in memory or kept on swap device as swapped out, disk block descriptors are of 2 types…

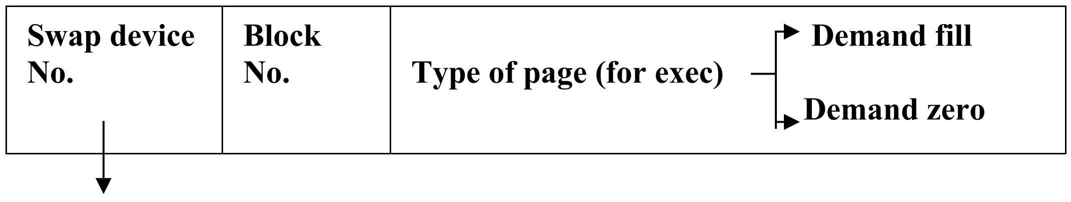
(*) **When the process is in memory :-** Disk block descriptor will have reference of hard disk block where the page of executable file of this process is located.

(*) **When the process is in swap device :-** Disk block descriptor will now reference to swap device block because now the page is located on the swap device as the process is swapped out to the swap device. So…

- **If the page is in executable file**, the disk block descriptor entry will contain the logical block number in the file that contains the page. As seen in file system algorithms, kernel can convert logical block number into the disk address.

- **If the page is in swap device**, the disk block descriptor entry will contain "LOGICAL DEVIC E NUMBER – BLOCK NUMBER" combination where the contents of the required page are located.

Single "DISK BLOCK DESCRIPTOR TABLE" entry



<u>This field will not be used if page is in executable file.</u>

By above figure now we can see the contents of disk block descriptor table entry – There are 3 contents

I. **Swap device number :-** This field will contain the logical device number of swap device if page is located on swap device. If page is located in the executable file, this field will be empty.
II. **Block number :-** If the page is located on swap device, this will contain the logical block number in which the contents of the required page are located. If the page is located in executable file, then this field will contain logical block address of the disk block in file where the page is located.
III. **Type of page :-** This field is only used for special conditions which are set during exec() system call. Here the page is of type either "DEMAND FILL" or "DEMAND ZERO".

Kernel allocates memory for above data structure dynamically.

(3) <u>Page Frame Data Table (or pfdata table) Entry</u> : - The pfdata table entry has "DESCRIPTION ABOUT ACTUAL PHYSICAL MEMORY PAGE". This table is indexed by the page number. Means information about a page can be obtained by using page number as an index which will gives respective page's pfdata table entry. The contents of each entry are….

A. **Page table :-** This indicates whether the required page is on swap device or is in executable file. Or alternately is also can indicate whether this page is re-assignable or not.
B. **Reference count :-** This indicates how many processes are referencing this page currently. Obviously this number will equal to the number of valid page table entries which are right now referencing this page. But for "SHARED REGIONS" it may differ with concerned to number of processes.
C. **The logical device number and block number :-** The same contents as of first two fields of disk block descriptor table. (why this field is duplicated in two data structures i.e. disk block descriptor table and the pfdata table? the reason will get cleared later).
D. **Pointers to other pfdata table entries :-** Which are on "FREE LIST OF PAGES" and on "HASH QUEUE OF PAGES" about this list and queue we will see soon.

Kernel allocates memory for this data structure only once for the lifetime of the system.

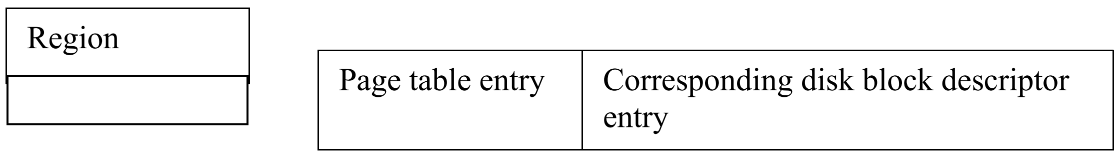
(4) <u>Swap-use Table Entry</u> :- This table contains an entry for every page on swap device, each entry contains a reference count of how many "PAGE TABLE ENTRIES" points to this page on swap device.

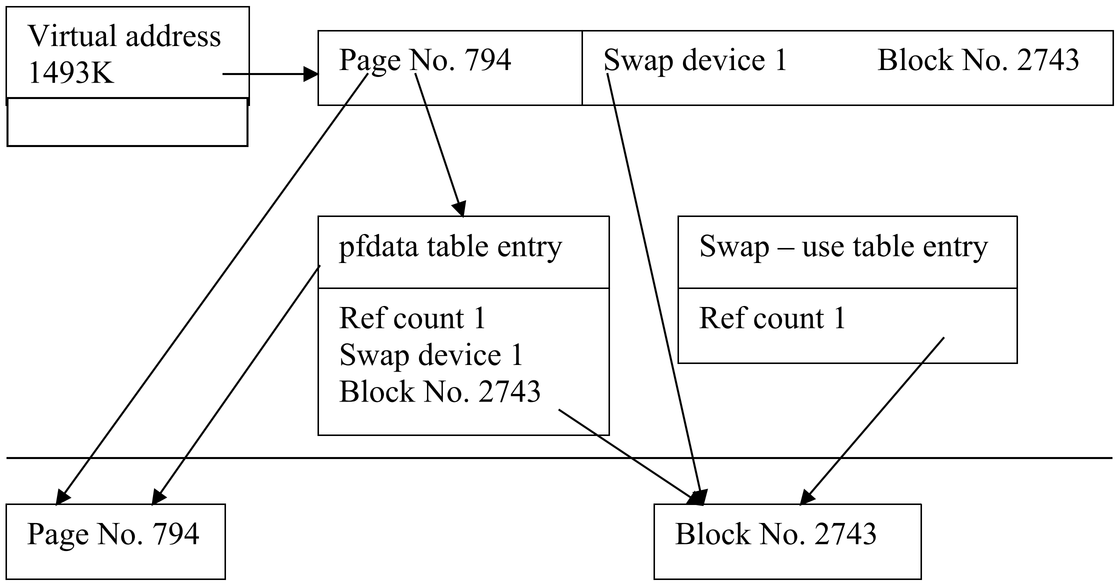
<u>Manipulation Of Above 4 Data Structures</u> :- As like for buffer, kernel also maintains "FREE LIST OF PAGES" and "HASH QUEUE OF PAGES". Similarly to buffer manipulation, the "FREE LIST" has such pages which can be re-assigned. But a process may fault on an address and can still find the corresponding page intact on the free list (this differs from buffer mechanism). If page is just used and then kept on free list, then kernel may found the page quickly from the free list when the same page is referenced quickly again after its release. This saves a read operation from file system or from swap device for this page. As like buffers kernel keeps released pages on free list in "LEAST RECENTLY USED" (LRU) fashion, and picks up the page at head of the list when required. So this is about a "RELEASED OR FREE PAGE".

Now when a page is assigned and contain valid data, it is kept on correct "HASH QUEUE OF PAGES". The hashing function is of logical device – block number combination same like buffer. (here logical device number is of swap device when swapping is concerned). So if device number and block number are given, kernel can locate a page quickly in memory (i.e. in hash queue of pages).

Kernel links pfdata table entries with the free list and hash queue of pages. The hashing of pfdata table entry with the hash queue uses the same "HASHING FUNCTION" as explained above.

So when kernel wants to assign a new page to a region, it gives to "FREE LIST", removes first free page (i.e. near most to header) from the list (i.e. from the "FREE PAGE FRAME ENTRY" in pfdata table), updates swap device accordingly for block numbers and finally puts the allocated page on correct hash queue by using the "HASHING FUNCTION".
Figure :- Following figure shows the relationship between
- Page table entries.
- Disk block descriptor table entries.
- pfdata table entries and
- Swap – use table entries.





Actual physical page **Actual swap device**

Relationship between 1493K on a region requires a new page.

- Virtual address 1493K on a region requires a new page.
- It is given a page of number 794 which is then referenced by arrow from virtual address to the page table entry of this page.
- For this new page, a page table entry is created and its 7 fields are initialized.
- Then corresponding disk block descriptor table entry is created. Assuming that page contents are now located on swap device, the swap device No. and block No. are filled into the disk block descriptor table entry, i.e. swap device 1 and block No. 2743 respectively .
- A corresponding pfdata table entry is created. Whose reference count field isinitialized to 1 and swap device No. and block No. are duplicated here.
- The swap-use table entry (as page is now on swap device) shows its reference count field 1 means one page table entry is pointing to page.

<u>Fork In Paging System</u> :- As seen in the explanation of fork() system call, kernel duplicates every region of the parent process into the child process during fork() system call end then attaches these duplicated regions to the child process.

When fork() is called in "SWAPPING" supported system . kernel makes physical copy of the parent address space. But if you lock at this from the view of demand paging, you will find that this is "UN-NECESSARY WASTE OF SYSTEM MEMORY". Because as usually fork() is immediately followed by the exec(), this physical copy of memory is going to be freed immediately and then replaced by program's (which is called by exec()) virtual addresses.

On "PAGING" supporting system kernel avoid this un-necessary copy of physical memory of parent . For this purpose kernel just increment the region reference count of shared regions. For non-shared i.e. private regions (mostly Data and Stack regions), it allocates a new region table entry and a new page table entry for that new region table entry.

Now kernel starts examination of each page table entry of parent's process. If a page is valid, it increments reference count field in pfdata table entry which indicates that now two processes (i.e. parent and child) are sharing same page via different regions (here "DIFFERENT REGIONS" means parent's private data and stack and child's private Data and Stack). So this is not the sharing of page by sharing the region. Rather this is the sharing of page by sharing same pfdata table entry. If the page, instead of existing in memory, if exists on swap device then kernel will increment 2 reference counts – one of pfdata table entry (as above) and of swap-use table entry.

Now the same page can be referenced through both regions (i.e. parent's & child's), which share this page until any of these 2 processes write on it.

Then kernel copies the page so that each region will have its own private page. To do this kernel turns "ON" the "COPY ON WRITE" bit for every page table entry in private regions of the parent and child processes during fork(). So that if any of these two processes tries to write on it, there will be "PROTECTION FALT", and while handling this fault, kernel will make a new copy of the page for that process (i.e. either parent or child) which is causing the fault. By this way physical copying is delayed as possible as until a process really want the page for writing.

Following figure shows data structures when a parent forks n child. Both processes have a shared region T, hence its reference count is 2. So the page 967 for virtual address 24K is also shared. But pfdata entry for this page shows reference count 1. Because though page is shared by 2 processes, the sharing is though the "SHARED REGION" and not though the "SHARED pfdata ENTRY".

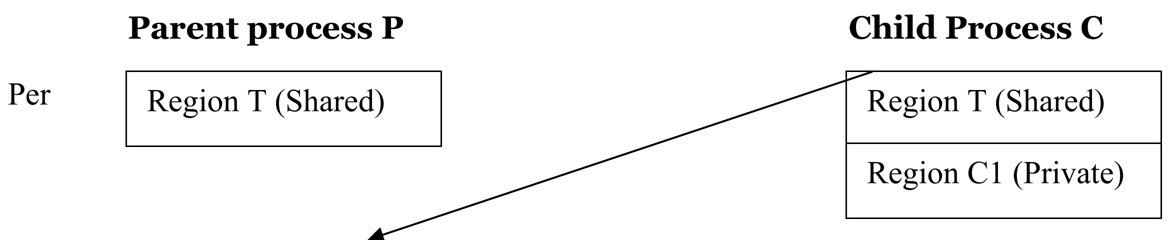
Now there is a private region of parent (say p1) which may be data or stack. As fork creates exact copy of this region in the child, but it is private to child, hence we will call it as C1. As they both i.e. P1 and C1 are identical, both will have virtual address 97K and both will reference to page 613. As both are private their reference count will be 1.

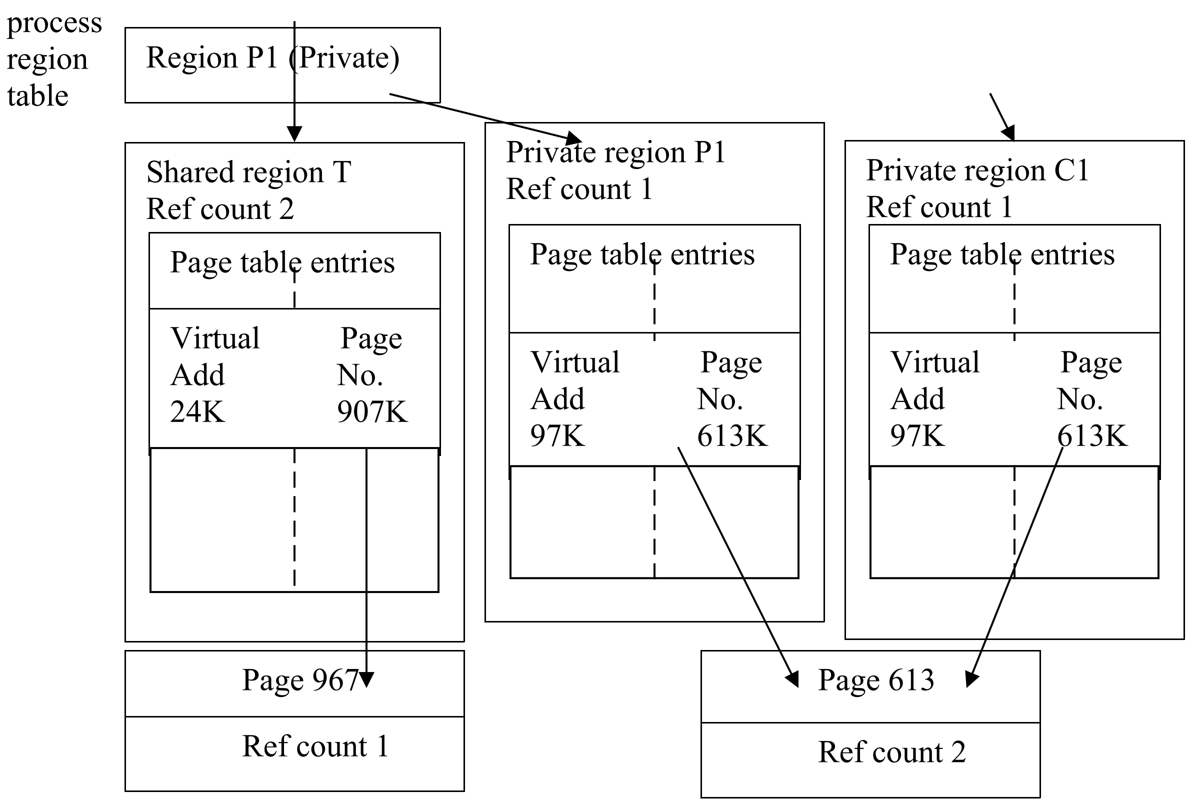
Now here again both processes are sharing one page, but there is no such reference count in their regions which will indicate 2. So here instead of regions, the pfdata table entry for page 613 will show its reference count 2. So this is the method of sharing common page by two processes when the regions are not shared but are private and still both private regions can share one single page.

If this page is in memory or in executable file (i.e. in file system), then data structures will be exactly as above. But if the page is in swap device, then to above data structures one will get added and that is swap-use table, whose entry will also show its reference count field incremented to 2. (as it is a referenced by two page table entries one P1 and one in C1).

The ***Free BSD*** system keeps 2 versions of fork. Where the ordinary fork() makes physical copy of parent process in child (as like old fork() in "SWAPPING" supported system).

But it also has vfork() system call which is made for such a situation where child will immediately call exec() after vfork(). In this, physical copying of pages is not done but child executes in same physical address space as that of the parent. This speeds up the performance of BSD. But creates a serious problems that, now child can overwrite parent's data (as child runs in physical address space of parent). So dangerous situation may arises if a programmer uses vfork() incorrectly. Thus advantage of vfork() is only for good programmers. ***vfork() also waits the parent, until child exec or exit***.





pfdata table entry for shared region and shared page **pfdata table entry for private region but shared page**

For example consider following program...

```
int global ; /* Obviously automatically initialized to 0 */
void main(void)
{
        int local;
        local  = 1;

        if(vfork() == 0)
        {
                /* child's code */

                  global = 2;                          /* writing on parent's data */
                local = 3;                          /* writing on parent's stack */
                - exit();
        }
                /* parent's code */
                printf("global = %d AND local=%d\n", global, local);
}
```

- Parent has one global variable "GLOBAL" whose value is 0. It has a local variable "LOCAL" whose value is 1.
- By executing vfork(), parent creates a child. Now assuming that this programmer is not good, he will not call exec() immediately (this is requirement of vfork), instead he will change values of "GLOBAL" and "LOCAL" variables of parent by overwriting its one of the page in data region

("GLOBAL" lies in data region) and one of the page in stack region ("LOCAL" lies in stack region). So when child exits.

- vfork() suspends parent's execution until child completes its exec() or exit (), parent will wait.
- Now when child exits by – exit() function call, parent will execute last printf() . expecting that the output would be global=0 AND local = 1, but gets the out global = 2 AND local = 3. Because child overwrote on parent's physical address space.

  Note :- For this program, instead of using exit() system call – exit() is used because exit() system call when used with vfork(), "CLEANS-UP" standard I/O data structures (we need this I/O in last printf) for both parent and child (i.e. flushes stdin and stdout). So this is another side effect of vfork(). To avoid this and to get last printf() correctly – exit() is used rather than exit().

<u>Exec in Paging System</u> :- When a process executes exec() system call, kernel reads the executable file from the file system (see algorithm of exec ()) into the memory.

Here exec() in paging system can work in 2 modes –

- Demand paging.
- Direct paging.

(1) <u>Exec in Demand Paging</u> :- executable file may be too large to fit in available memory at once. Here exec works in demand paging mode.

- Kernel therefore first assigns page tables and disk block descriptor for the executable file.
- While creating disk block descriptor table entries, it marks the "TYPE Of PAGE" field with "DEMAND FILL" for non bss data and "DEMAND ZERO" for bss data.
- By using a special version of read() algorithm, now it starts reading executable file into the memory.
- But such a scheme is used that, process will give validity fault on each page it is reading.
- The validity fault handler (which we will see later) notes whether the page (for which fault occurred) is "DEMAND FILL" or "DEMAND ZERO".
- If the page is "DEMAND FILL", the contents of page will get immediately overwritten with the contents of executable file. Means it does not need clearing first and then writing on it.But if the page is "DEMAND ZERO", then its contents should be first cleared and then writing contents of executable file should be done.
- If process can not fit in memory, the page stealer process will periodically swaps pages from memory to swap device to make more room for further contents of executable file.

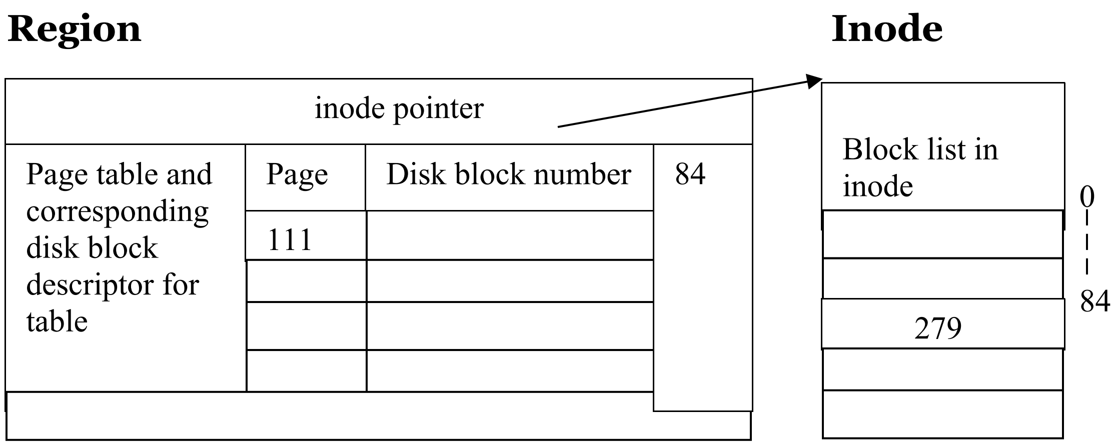
<u>Process in above scheme</u> :- There are many problems in "DEMAND PAGING" scheme for exec().

1. As validity fault is going to occur on each page, it is also going to occur on those pages which process never accessed.
2. The page stealer process, as it is periodical (not event based), is going to execute surely. So such thing may happen that, page stealer process arrives to swap a page but exec of that page is not done yet. This result in two swap operations un-necessarily, one when page stealer swaps out the page, then it is demanded again and after its use it will be again swapped out.
3. To avoid above problem, (i.e. sending the page to swap device by page stealer process and again making "SWAP IN" from swap device) the kernel can demand the page directly from executable file (provided the data on executable file is properly aligned. This is given by magic number). But beginning page from executable file, which is now in file system, requires use of standard file I/O algorithms such as bmap. bmap() may be expensive if page is to be mapped from indirect blocks.(recall that bmap internally uses bread() and brelease() multiple times in a "WHILE LOOP"). There is one another problem with bmap is that, it is not re-entrant from the point where we left it previously. So it has to be called again, starting its all tasks right from the beginning, though not needed here for one file.

4. For ordinary read() system calls, kernel sets many I/O concerned parameters in process's U area. Now if this ordinary read() is used for "PAGE FAULT" generating situations in "exec IN DEMAND PAGING", then "OVERWRITING OF U AREA CONTENTS" will take place. So kernel can not use ordinary read() system call. Instead it has to use a special variation of read() system call as mentioned at the beginning of this topic.

(2) Exec in Direct Paging :- When executable file's size can fit into the available memory, then this mode is used and it is quite simpler and more "PROBLEM FREE" than "EXEC IN DEMAND PAGING MODE".

- Here kernel finds out all disk block numbers of the executable file during exec and accordingly it attaches this list to the file's inode.
- While setting page table entries and disk block descriptor table entries it writes above block numbers in disk block descriptor table corresponding to the page in page table entry. (disk block number start from 0).
- Later validity fault handler uses this information in disk block descriptor entries to load the required page from respective block.)
- New when a page is required, disk block descriptor table gives block number of the page, then the region (in which this page referenced) has inode pointer, from which inode is accessed and then in that inode above disk block is searched and read (via 13 member array of disk blocks) into the memory.



Suppose page 111 is required, its corresponding disk block descriptor table entry gives block number (logical) as 84. Then from region's inode pointer field, file's inode is accessed and data block 279 is searched and read which is at 84th index in disk block array in inode. Now data block 279 has contents of page 111.

The Page Stealer Process :- It is of the type "KERNEL ONLY" process. The function of this process is to "SWAP OUT" memory pages which are now no longer part of the working set of a process.

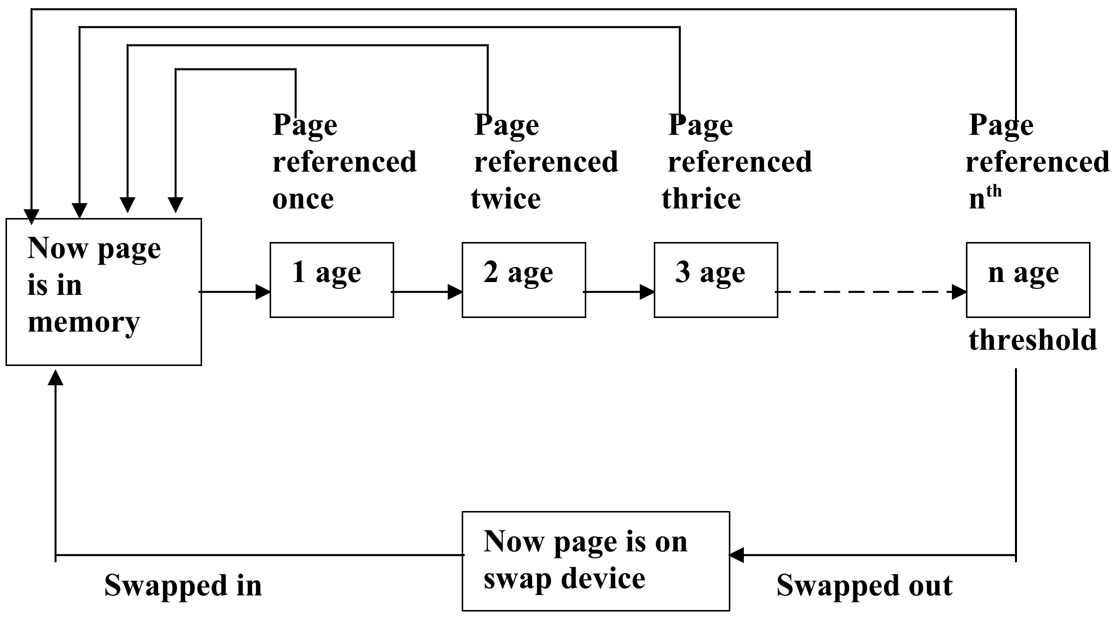
Kernel creates page stealer process during system initialization and invokes it continuously time to time throughout the life time of system, when system's memory is low on free pages.

So during its life time, stealer process examines every active but unlocked region and skips only locked regions. Means these locked regions will be examined in next execution of this stealer process if they get unlocked meanwhile.

During its each execution of unlocked but active regions it increments age field of page table entry of every page which is valid (i.e. contents of page are valid).

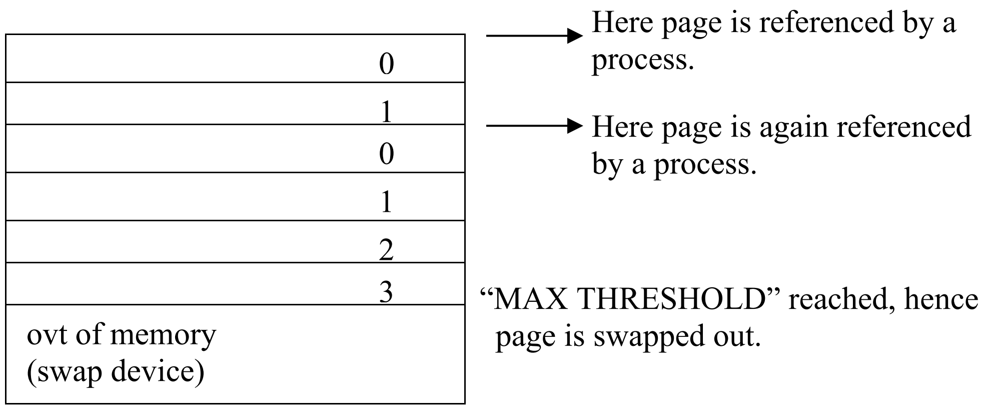
How kernel protects examination of a page which is being used ? when a process faults on a page, then it is indication that this page is going to be used immediately. Now we know that pages are referenced in region. So when a page is going to be used immediately, it locks that region which contains this page, preventing stealer process to steal that page on which fault had occurred.

States :- In memory every page is present either in one of the two states…
(a) The page is still undergoing "AGING PROCESS" and hence yet not become eligible to "SWAP OUT".
(b) The page is completed its "AGING PROCESS" and thus becomes eligible for "SWAP OUT" and also become eligible for reassignment to some other virtual address than the previous.
The (a) state indicates that the page is recently accessed by a process and thus it is a part of working set of that process. Some machine hardware can set "REFERENCE BIT" part of page table entry of such page when it is accessed. But if hardware does not support this, then kernel can set it via software interface. When page stealer gets executed, during its examination of above page, it turns "OFF" that "SET REFERENCE BIT" but remembers "HOW MANY EXAMINATIONS IT HAD DONE WITH THIS PAGE, SINCE LAST REFERENCED BY PROCESS". So on each examination page goes on aging (we can say one examination by stealer process of the page is "ONE AGE OF THE PAGE", then "TWO AGE OF THE PAGE", then **"THREE AGE OF THE PAGE"** and so on). There is a "DEFAULT THRESHOLD NUMBER OF AGE OF A PAGE", when this number is exceeded by the page, the page enters in second stage and becomes eligible for swapping out-state (b).



The maximum period of "AGING" of page (so it can become able to get swapped out) is thus implementation dependent, and it also has a constraint of available number of bits in the page table entry (i.e. age field).
Following figure shows the instruction between "PROCESS ACCESSING A PAGE" and "EXAMINATION OF THAT PAGE BY STEALER PROCESS".

Page state	**Time (last referenced)**
In memory	0
	1
	2


0
1
0
1
2
3
ovt of memory
(swap device)
Here page is referenced by a process.
Here page is again referenced by a process.
"MAX THRESHOLD" reached, hence page is swapped out.

The page starts its life in main memory (as it is first accesses by some process) and its aging starts. When stealer process examines it for the first time its age is 1, then during second examination age becomes 2. At this time it is again referenced by some process and hence age drops to 0 (from whatever its current age). Then its life again starts from 0, it is again examined by stealer process so age becomes 1. Here the page gets referenced by some process and hence again age becomes 0. After 3 examinations by stealer process, yet no process has referenced the page, "MAX PAGE THRESHOLD" is reached (we assumed that threshold = 3 although actually it is implementation dependent) and thus stealer process swap it out from main memory to swap device.

If two or more processes share a region, they update the "REFERENCED BIT" field of the page table entries of every page in that region. So a page can be the part of working set of more than one process. But this does not matter to the stealer process. Stealer process always follows a simple rule…"IF A PAGE IS PART OF WORKING SET OF ANY PROCESS, IT WILL REMAIN IN MEMORY AND IF IT IS NOT PART OF THE WORKING SET OF ANY PROCESS, IT IS ELIGIBLE FOR SWAPPING". Number of pages in a region does not matter anything to stealer process and stealer process does not try to swap out equal number of pages from active, unlocked regions.

<u>How stealer process works</u> :- Kernel keeps 2 "LEVEL MARKS" low and high for the memory. If the free available memory is below "LOW LEVEL MARK", it wakes up the stealer process and stealer process starts swapping out the pages making more and more free memory. When available free memory amount rises above the "HIGH LEVEL MARK", page stealer process stops "SWAPPING OUT" of pages. Administrator can set these 2 marks for desired best system performance.

When page stealer process decides to "SWAP OUT" a page, it looks for the presence of copy that page on swap device. Here 3 possibilities are there…

(a) There is no copy of this page on swap device but kernel had scheduled this page for swapping
    - In such case page stealer process just keeps number of (not the actual page) this page no "SWAP PAGE LIST" and continues. When this list reaches its maximum count, kernel writes these pages on the swap device (whether all ? or some of them ? – answer is coming soon)

(b) There is a copy, already present on swap device, and no process had yet modified this page's "MEMORY CONTENTS" (i.e. in core contents)
    - In this case kernel clears "VALID BIT" of this page in its page table entry, then decrements its reference count in its pfdata entry and puts this pfdata entry on the "FREE LIST" for possible future allocation.

(c) There is a copy, already presents on swap device, but a process had modified its "MEMORY CONTENTS" (i.e. in core contents).
    - Here kernel schedules the page for "SWAPPING" by telling the stealer process to keep this page on "SWAP PAGE LIST" (as explained in case (a)). As contents of this page are overwritten, its copy on "SWAP DEVICE" is useless. Hence kernel frees its entry on swap device making room on swap device. Page stealer process swaps the page to swap device in case (a) and in case (c).

The difference between case (b) and (c) is that, in case (b) as "IN CORE CONTENTS" are not yet modified copy of page in memory and on swap device are identical. Hence there is no need of swapping the page on swap device as it is already there. But if a process changes in core contents of the page, then its "MEMORY COPY" and "SWAP DEVICE COPY" differ and thus it is must for kernel to write the page on swap device after freeing its previous entry on swap device. Obviously this writing will be in new entry of swap device because it does not use same, previously freed entry of this page for new writing. By this way kernel maintains contiguity of entries on swap device.

Difference and similarity between "SWAPPING OUT" of page by "SWAPPING POLICY" and "PAGING POLICY" :-

- ✓ In "SWAPPING POLICY", every page is "SWAPPED OUT" (which belongs to a process) one by one.
- ✓ In "PAGING POLICY", page stealer process fills a "SWAP PAGE LIST" of eligible pages and "SWAPS OUT" only when this list gets full. So multiple pages are swapped out at once. The only condition for which the page is kept on list is "COMPLETION OF AGING" of that page. But similarly is that, in both policies, the method of writing of "SWAPPED OUT" page to swap device is identical(see **"SWAPPING PROCESS OUT"**, **"FORK SWAP"** & **"EXPANSION SWAP"** on **pages 236 to Page 241**). But there is one catch ! if swap device / devices do not contain enough continuous pages, than in paging system too, kernel swaps one page at a time even page list is full. This is obviously too slow.

  As in "SWAPPING POLICY", the "SWAPPING OUT" and "SWAPPING IN" of pages takes place one at a time, there is less fragmentation of swap device. But in "PAGING POLICY", "SWAPPING OUT" takes place with "BULCK OF PAGES" and "SWAPPING IN" takes place "ONE AT A TIME".

  Hence there is more and more fragmentation of swap device.

How kernel writes pages to swap device ? :-

- ✓ When kernel writes a page to swap device, it turns "VALID BIT" of page tale entry of that page "OFF".
- ✓ It decrements "REFERENCE COUNT" field in pfdata table entry of this page.
- ✓ If this count drops to 0, it places this entry on "FREE LIST" at end. (Similarly to buffer algorithm).
- ✓ If this count is not 0, then it is an indication that, more processes are sharing this page (as seen in "fork() IN PAGING SYSTEM") but kernel still swaps the page.
- ✓ Finally kernel allocates space for this page on swap device, saves the swap address (i.e. address of this newly allocated space on swap device) in disk block descriptor table entry of this page and thus increments "REFERENCE COUNT" field in "SWAP USE TABLE ENTRY" of this page (as page is now going to reside on swap device).

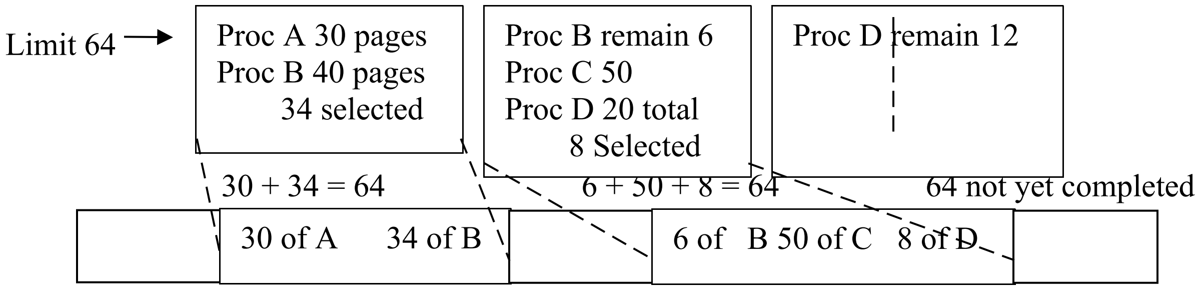
  Here a question may arise, what will happen if process gives fault on this page when the page is in "SWAP PAGE LIST" ? Though entry of the page on "SWAP PAGE LIST" indicates that the page is eligible for "SWAP OUT" and soon will be, if now a process gives fault on this page before page is written to swap device, then kernel allows process to access this page from "SWAP PAGE LIST", instead of getting it back from swap device (so reduces one disk I/O).

  But **note that** still page is "SWAPPED OUT" (as it gets truly aged and eligible) if page is still on the "SWAP PAGE LIST". (This will happen when process gives fault for this page just for reading and not for writing on it).

Example :- Suppose page stealer process has swapped out 30, 40, 50, and 20 pages from processes A, B, C, And D respectively. Also suppose that stealer process is able to write 64 pages at once in one swap disk write operation. Then following figure shows the sequence of swapped out page writing method on swap device.

In other words . the "SWAP PAGE LIST" limit is of 64 pages.

Total swapped pages from A, B, C, And D.



As page stealer can write 64 pages to swap device at one time (as assumed), it allocates space for 64 pages on swap device and swaps out all 30 pages of process A but swaps out 34 pages of process B (out of 40) as 64 quota gets complete.

Then it allocates another space for 64 pages on swap device (Note that first space and this space need not to be continuous . Hence gap between those two spaces is shown in figure above) and swaps out remaining 6 pages of process B, then all 50 pages of process C, but 8 pages of process D (out of 20) as 64 quota is completed.

Now remaining 12 pages of process D are kept on "SWAP PAGE LIST" for next operation because yet 64 quota is not completed. (means list is yet not full).

<u>How free space develops on swap device</u> ? :- If a page fault occurs, means a page is needed to a process, and now page is on swap device, then its entry is removed from swap device making free space on swap device.

Also when a process exits and if some of its pages are on swap device, then as exited process no longer going to make fault for these pages, all pages concerned only to that exited process are removed from the swap device, making free space on swap device.

Contents of a page are said to be valid, until this page is not reassigned to any process's virtual address space.

<u>Page faults</u> :- The system can give 2 types of page faults.
(1) Validity fault and
(2) Protection fault.

The functions or algorithms which handles these faults are thus called as fault handlers.

During the process of fault handling, the fault handler may need to read the page either from the disk file system (i.e. from executable file) or from swap device and then pulls this page into memory. So during this I/O, the fault handler may sleep. Usually interrupt handlers does not sleep, but "FAULT HANDLER" is an exception to this rule. One thing is better that, fault handler sleeps in the context of that process which had made the fault, thus fault is related only to that currently running process. Thus only this process gets asleep preventing sleeping of arbitrary processes.

(1) <u>Validity Fault Handler</u> :- When a process tries to access such a memory page whose "VALIDITY BIT" is not set, then process incurs a validity fault and kernel then invokes "VALIDITY FAULT HANDLER" function or algorithm.
Which are the pages whose valid bit is not set ?
(a) Pages outside the virtual address space of process.
(b) Pages which are part of virtual address space of the process but yet not assigned to the actual physical pages.

The algorithm for “VALIDITY FAULT HANDLER” is like follows…

```
01 : algorithm : vfault() /* validity fault handler */
02 : input : address, where the process faulted.
03 : {
04 :      Find (a) region of the address
05 :           (b) page table entry of that address.
06 :           (c) Disk block descriptor table entry of address.
07 :        Lock this region;
08 :        if(the given address is outside the virtual address space of the process)
09 :              {
10 :                    send signal(SIGSEGV : segmentation violation) to the process;
11 :                    goto out;
12 :              }
13 :        if(uptill now address becomes valid) /* As process may sleep above */
14 :              goto out;
15 :        if(page belonging to address is in “page pool” i.e. cache)
16 :              {
17 :                    remove page from cache;
18 :                    adjust page table entry accordingly;
19 :              while(page contents are not valid)
                    /* because some other process already using page */
20 :                    {
21 :                          sleep(event : until page contents become valid);
22 :                    }
23 :      }
24 :     else /* means page is not in “pool” (i.e. cache) */
25 :     {
26 :      assign new page to the region;
27 :      put this new page in cache;
28 :      update pfdata tale entry accordingly;
29 :      if(page is not loaded previously and page is set “demand zero”)
30 :              {
31 :                    clear this new page and initialize it to 0;
32 :              }
33 :      else /* means page is either loaded once previously or it is
                    “demand fill” */
34 :              {
35 :                    read virtual page because it is already present somewhere either from
        disk
                          file system (i.e. from executable file) or from swap device;
36 :                    sleep(event : until this I/O is not done);
37 :                    }
38 :                    awake all processes which slept on the event of becoming page
contents valid;
39 :      }
40 :      set page’s valid bit;
41 :      clear page’s modify bit;
42 :      clear page’s age bits;
43 :      recalculate process’s priority;
44 :    out : check for any pending signals, unlock region, which was locked at the beginning of this
            algorithm;
45 : }
46 : output : nothing
```

1. When process accesses such an address whose page's valid bit is not set, this address is passed to kernel.
2. Kernel invokes "PAGE FAULT HANDLER" algorithm.
3. In this algorithm, kernel first finds out the region, page table entry and disk block descriptor table entry corresponding to this address.
4. Kernel then locks this region prevents "RACE CONDITION", which may occur if stealer process tries to swap out this page.

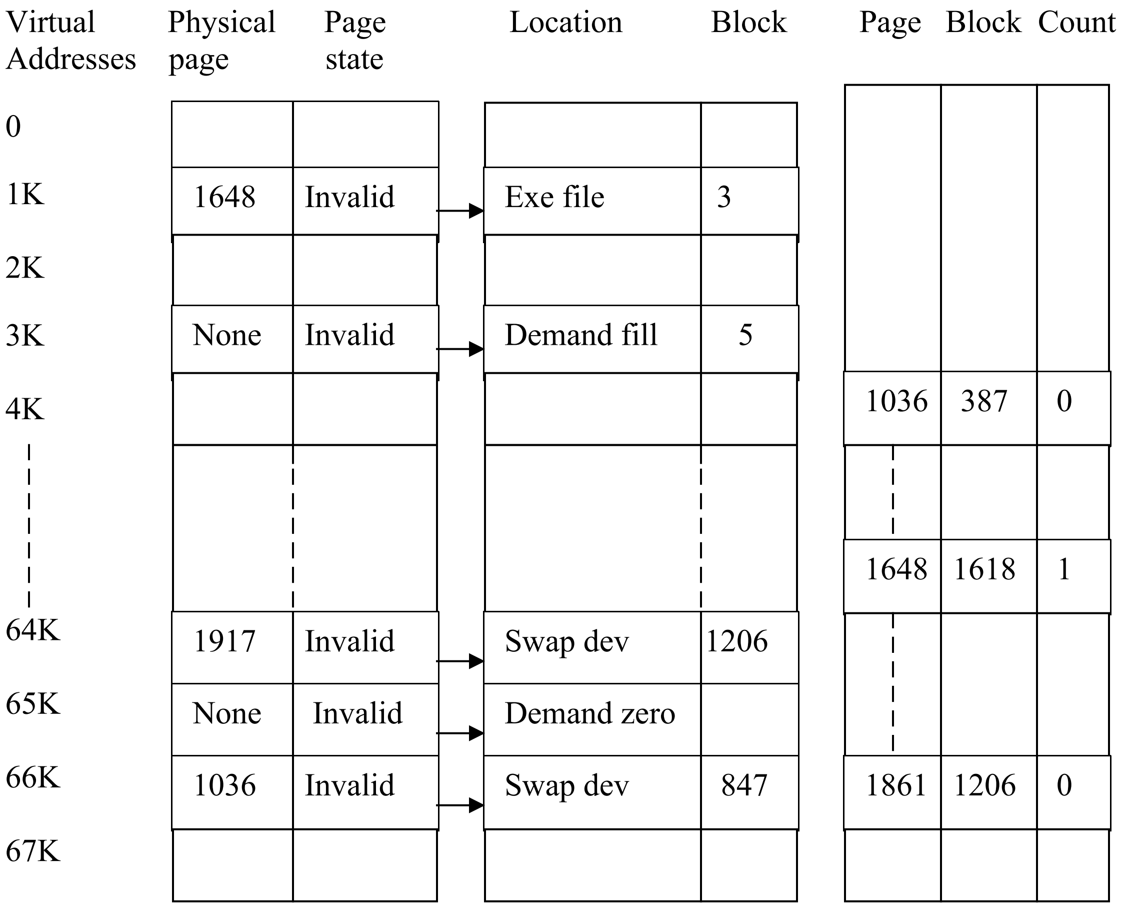
- Sometimes it may happen that the address accesses by the process is outside of its virtual address space, means reference to the accessed memory address is invalid, hence kernel stops further invocation of algorithm and sends signal to the process of "SEGMENTATION VIOLATION", unlocks the locked region and returns.
- But if the address accessed by the process is within the virtual address space of the process (i.e. memory referenced by the process is valid) then the memory page corresponding to this address is located at one of the following 5 locations and accordingly page may be in one of the following 5 states..
  (1) Page is on swap device and not in memory.
  (2) Page is on "LIST OF FREE PAGES" in memory.
  (3) Page is in process's executable file.
  (4) Page is marked as "DEMAND ZERO".
  (5) Page is marked as "DEMAND FILL".
- **Case (1)** If the page is on swap device and not in memory, then this means that sometime previously page was in memory and page stealer swapped it out to the swap device.

  So from the "DISK BLOCK DESCRIPTOR TABLE" (which was found previously) entry, kernel finds out the swap device and the block number where the page is stored. Kernel also verifies that the page is really not in the page cache.

  Now kernel updates page table entry of this page in such a way that it will now point to the page which is going to be read now. Then kernel places pfdata table entry of this page on the "HASH LIST". So that further operations of fault handler become speedy. Then kernel starts reading of the page from the swap device into the memory.

  During this I/O the process which made the "FAULT", sleeps and when I/O completes, kernel awakens all the processes (waiting for this page to become valid) including this process.

Page table **Disk block descriptor table** **pfdata table**



Virtual Addresses	Physical page	Page state	Location	Block
0				
1K	1648	Invalid	Exe file	3
2K				
3K	None	Invalid	Demand fill	5
4K				
64K	1917	Invalid	Swap dev	1206
65K	None	Invalid	Demand zero	
66K	1036	Invalid	Swap dev	847
67K				

Page	Block	Count
1036	387	0
1648	1618	1
1861	1206	0

Consider the figure on last page for the “PAGE TABLE ENTRY” of virtual address 66K. Now if process incurs “VALIDITY FAULT” for physical page 1036 as it is “INVALID”, then “FAULT HANDLER” algorithm finds corresponding “DISK BLOCK DESCRIPTOR TABLE ENTRY”, where it finds that corresponding page is on swap device at 847 block. This means that virtual address is legal (in other words the virtual address accesses by process is in the process’s address space). Now algorithm searches “PAGE CACHE” i.e. pfdata table entries, but can not found entry for block 847 in it. Thus it is clear that page is not in memory but on swap device.

So now the page has to be read from swap device. Thus kernel assigns new page, say 1776, reads contents of virtual page on swap device at 847 (recall that pages on swap device are not real pages, but are virtual pages, and hence memory pages in swap device is so called the virtual memory.) into this new physical page and accordingly updates all the 3 tables......

Page table entry **Disk block descriptor table entry** **pfdata table entry**

Virtual Address	Physical page	Page state	Location	Block No.	Physical page	block no.	count
66K	1776	Valid	swap dev	847	1776	847	1

- ✓ For page table entry, it writes new physical page number i.e. 1776 in place of 1036. In page state it updates to “VALID” in place of “INVALID”.
- ✓ For disk block descriptor table entry, there is no change. Means the page has 2 copies, one virtual copy on swap device at block 847, and another, real or physical copy in page 1776.

- ✓ For pfdata table entry, previously there was no entry for this page. Now a new entry is added by writing 1776 in “PHYSICAL PAGE COLUMN”, block 847 in “BLOCK NUMBER COLUMN” and writing 1 in “COUNT” column indicating that this page is now referenced by one process, once.
- ➢ **Case (2)** Sometimes it may happen that the “DISK BLOCK DESCRIPTOR TABLE” shows that the page is on swap device (means swapped out), but it has entry in pfdata table entry. Means the page is in memory. On page 266, the figure shows this condition at virtual address 64K, where the block num 1206 of swap device has entry in pfdata table.

  Usually the physical page number in page table entry and pfdata table entry are same when same process accesses the page. But it may happen that some another process may faulted on same virtual page into different physical page. That is why the page table entry shows physical page number 1917 while pfdata table entry shows physical page number 1861.

  Above thing is one of the possibility. Means it is also possible that no other process faulted on same page and thus kernel never yet reassign new physical page.

  Coming back to our case 2, now we know that page is in memory. So no swap device I/O is needed. Thus kernel finds this page in memory (i.e. in page cache – pfdata table) by matching “BLOCK NUMBER FIELD” of “DISK BLOCK DESCRIPTOR TABLE” (eg. – 1206 in above example) with the “BLOCK NUMBER FIELD” in pfdata table.
  Then it simply updates page table entry by changing physical page number 1917 to 1816. so now the virtual address 64K is pointing to a valid page 1861, hence the “STATE” field in page table entry becomes “VALID”.

  As one process is now referenced this page, the reference count in pfdata table entry is incremented .

  Above two cases, shows the importance of keeping “BLOCK NUMBER FIELD”, both in disk block descriptor table and in pfdata table, though initially it looks like redundancy.
- ➢ If process A is faulted on a page, and finds that another process is already faulted on it but the reading of the page by another processes is still going on, then process A finds that the region corresponding to the required page is already locked by another process’s “FAULT HANDLER” and thus sleeps until another process’s fault handling gets completed. This is called as Double Fault on a page.
- ➢ **Case (3)** Sometimes the page is neither on swap device nor in memory but present in executable file of the process which is on disk in its file system.

  If we recall the figure on page 258, we come to know that disk block descriptor table entry for such a page will contain block number of this page. By this block number inode of the file is found which is associated with the region table entry of this page. Then this block number is used as offset into the inode of this executable file(i.e. in the 13 elemental array in inode), now finds the actual data block for this page and then reads its content into new physical page.

  As an example, we can look at virtual address 1K in figure on page 266, which shows in its disk block descriptor table entry, that, the page 1648 which is “INVALID”, is located in executable file at block number 3.
- ➢ **Case (4) & (5)** When a process faults on such a page, which is in the state, either “DEMAND ZERO” or “DEMAND FILL”, kernel then allocates free page from memory (taking from “FREE LIST OF PAGES”), updates its “PAGE TABLE ENTRY”. If “DEMAND ZERO”, it checks the page contents to 0 and then clears the “DEMAND ZERO” flag in disk block descriptor table. For “DEMAND FILL” it just clears the “DEMAND FILL” flag in disk block descriptor table entry of this page.

  The page is new valid. It is not on the swap device(i.e. not swapped) and also not on the file system (i.e. not in the executable file)

  This will happen when accessing a virtual addresses 3K & 65K in figure on page 266. as these pages do not have any pfdata table entry, their ref. Count is 0, means no process yet accesses those pages since this process is started.

- **The algorithm ends by doing 3 things …**
  1. It sets the “INVALID” page bit to “VALID”.
  2. It clears “MODIFY” bit end thus clears page age
  3. It recalculates this process’s priority. This is very important because process may slept in memory in kernel mode during “FAULT HANDLING”, which gives this process a greater priority than others and some processes may be waiting prior to current process for this page. So to give all processes “EQUAL CHANCE”, it recalculates this process’s priority.
  4. Then before referring to “USER MODE” it checks for any arrival of signals like SIGSEGV.
  5. Finally unlocks the region.

(2) <u>Protection fault Handler</u> :- Protection fault is a second type of memory page fault which occurs when “PROCESS ACCESSES A VALID PAGE, BUT THE PERMISSION BITS ASSOCIATED WITH THAT PAGE DO NOT PERMIT ACCESS FOR THE PROCESS TO THAT PAGE(IN OTHER WORDS PAGE IS PROTECTED)”. Secondly a process can also come across a protection fault, when “ IT TRIES TO WRITE ON SUCH A MEMORY PAGE WHOSE” copy on write “BIT IS SET”. In this second case, the kernel must find out whether really “COPY ON WRITE” is set and hence permission is denied or whether something else illegal has happened resulting in denial of the permission

The algorithm for “PROTECTION FAULT HANDER” is as follows …..

```
algorithm : pfault          /* protection fault handler */
 input : address, where the process faulted
{
 find (a.) region of the address
        (b) page table entry of the address
       (c) disk block descriptor table entry of address
 lock this region;
 if (page is not valid)
         goto out;
 if (“copy on write“ bit is not set)
         goto out;  /* real program error – denote signal */
 if (page’s pfdata entry’s ref count > 1)
 {
         allocate a new physical page;
         copy contents of original page to this new page;
         decrement original page’s pfdata table entry’s reference count;
         update page table entry accordingly to point to this new page;
 }
 else  /* means nobody is using this page as ref count is */
 {       /* - not greater than 1. hence get original page */
         if (copy of original page exists on swap device)
                 free space on swap device, break page association;
         if (page is on hash queue)
                 remove it from hash queue;
 }
 clear “copy on write” bit in page table entry, set modify bit;
 recalculate process priority;
 check for signals;
   out : unlocks the  region;
```

30: } output : nothing

- First 4 steps, as on page 265 are same as that of "VALIDITY FAULT HANDLER".
- The "PROTECTION FAULT HANDLER" has a pre-requisite, that the page must be valid. If it is not, the handler leaves the algorithm by unlocking the region. Checking of validity is must, because handler may sleep during the region is locked and thus stealer may swap out this page clearing its valid bit and making it invalid. In such cases "VALIDITY FAULT HANDLER" should run first before protection fault handler.
- Then it determines whether "COPY ON WRITE" bit of this valid page is set or not. If "NOT", then it assumes something really illegal is happened and leaves the algorithm by unlocking the region. Later kernel can determine the illegal thing by studying the signal sent to the process.
- If "COPY ON WRITE" bit is set, then it proceeds and checks whether the page is shared by some other processes. It checks this by looking at the page's pfdata table entry's reference count field if this is greater than 1, then the page is shared.

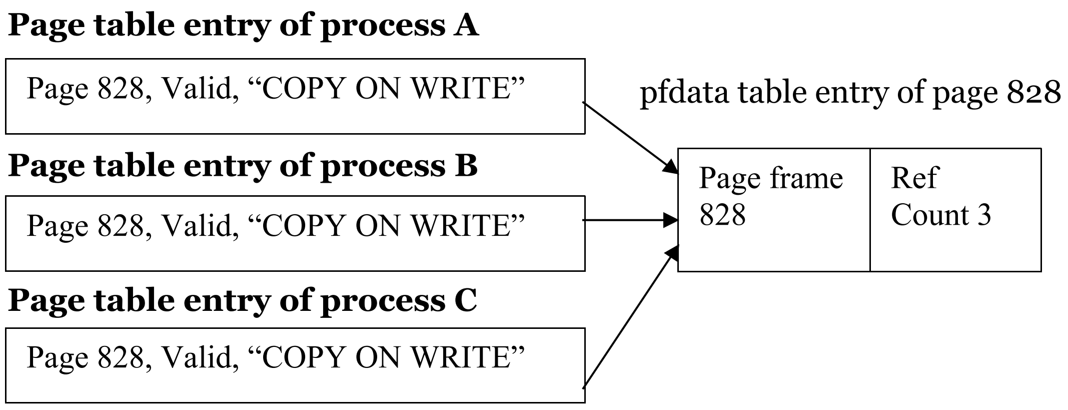
  So if the page is shared, it can not directly modify it. Thus a new physical page is assigned, then contents of the original shared page is copied onto this new page.

  Obviously this process is now out using the shared page but will use this new page. Hence reference count field in pfdata table entry of original page is decremented.

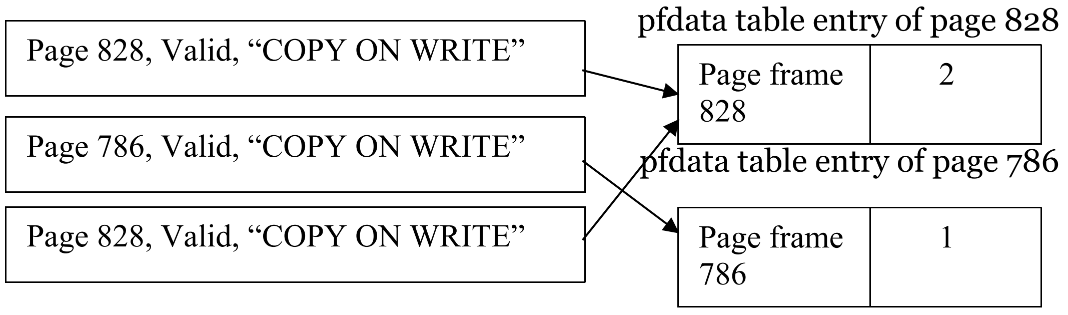
  As the virtual address(i.e. faulted address) is now not pointing to original page, its entry is removed and new page's entry is made in page table entry. So new virtual address will point to this new page.

  <u>Note that</u> **:- *Only this process will now reference to this new page. Other process will continue referencing to the original shared page.***

- (a) Before any process incurs protection fault on page 828.


Page table entry of process A
Page 828, Valid, "COPY ON WRITE"
Page table entry of process B
Page 828, Valid, "COPY ON WRITE"
Page table entry of process C
Page 828, Valid, "COPY ON WRITE"
pfdata table entry of page 828
Page frame 828
Ref Count 3

- (b) Assume that process B incurs protection fault on page 828 (After "PROTECTION FAULT HANDLER" runs for B)


Page 828, Valid, "COPY ON WRITE"
Page 786, Valid, "COPY ON WRITE"
Page 828, Valid, "COPY ON WRITE"
pfdata table entry of page 828
Page frame 828
2
pfdata table entry of page 786
Page frame 786
1

Suppose 3 processes A, B, & C share a common valid page of number 828, and for all these 3 processes, “COPY ON WRITE” bit is set in their respective page table entries. Obviously the pfdata table entry will show reference count 3.

Now suppose process B tries to write on this page. As “COPY ON WRITE” bit is set, it comes across a “PROTECTION FAULT”. So the protection fault handler will run for B and will allocate a new page 786 (as 828 is shared), copy contents of 828 on 786, decrements 828’s reference count to 2(from 3) and increments 786’s reference count to 1. it also updates page table entry of B, by wiping off 828, replacing it by 786 and clear “COPY ON WRITE” bit for 786.

- If the page is not shared, means if page’s pfdata table entry’s reference count is not greater than 1(it will be 1 because this process is accessing the page), then kernel allows this process to reuse this physical page. Though “COPY ON WRITE” bit is set initially, as this process now can reuse this page, the bit is cleared (as if it is not there).

  Now if the page is on swap device(as it was previously swapped out), it frees this page’s entry on swap device and disassociates this page from its swap device copy by updating the disk block descriptor table entry & swap use table entry. Then it uses this page. Decrement count if count 0, frees.

  If the page is in page hash queue, then it removes entry of this page from hash queue and then uses this page.
- The final steps are same as that of “VALIDITY FAULT HANDLER”. On of Page 268(bottom) and Page 269

After freeing of this page’s entry on swap device, the pfdata table entry is also removed, because page is not on swap device virtually.