CHAPTER 6 THE STRUCTURE OF PROCESS

In chapter 2, we saw "process" briefly. In this & the following chapters we will study "THE PROCESS" more thoroughly.

* **Process States And Transitions:-**

As stated in chapter 1, "Files have places & process has lives." So life time of a process can be divided into different states.

There are such (9) states...

1) The process is executing in user mode.
2) The process is executing in kernel mode.
3) The process is currently not executing but it is in "READY TO RUN" state as soon as the kernel schedules it.
4) The process is sleeping but it is present in main memory.
5) The process is "READY TO RUN", but the swapper process (i.e. process 0)
6) should pull it into main memory before kernel schedules the process to execute.
7) The process is sleeping, but the swapper process (i.e. process 0) has swapped the process to secondary storage(means the slept process is not in main memory) to make room in the main memory for other processes to run.
8) The process is returning from kernel mode to user mode, but the kernel
9) preempts it and does a "CONTEXT SWITCH" to schedule some other process.
10) The process is just newly created, so it is in such a transition state that, process exists but it is neither in "READY TO RUN" nor in sleeping state. This state is the "STARTING STATE" for all processes, except process 0.
11) At the termination time, process executed exit() system call and it is thus in zombie state. The zombie state is the signal state of a process. In this state process is "NO LONGER EXIST" but some traces of it are left behind, such as...

- A record containing exit code of the process and
- Some time related statistical data for the parent process.

Out of these (9) states first (4) are seen briefly before.

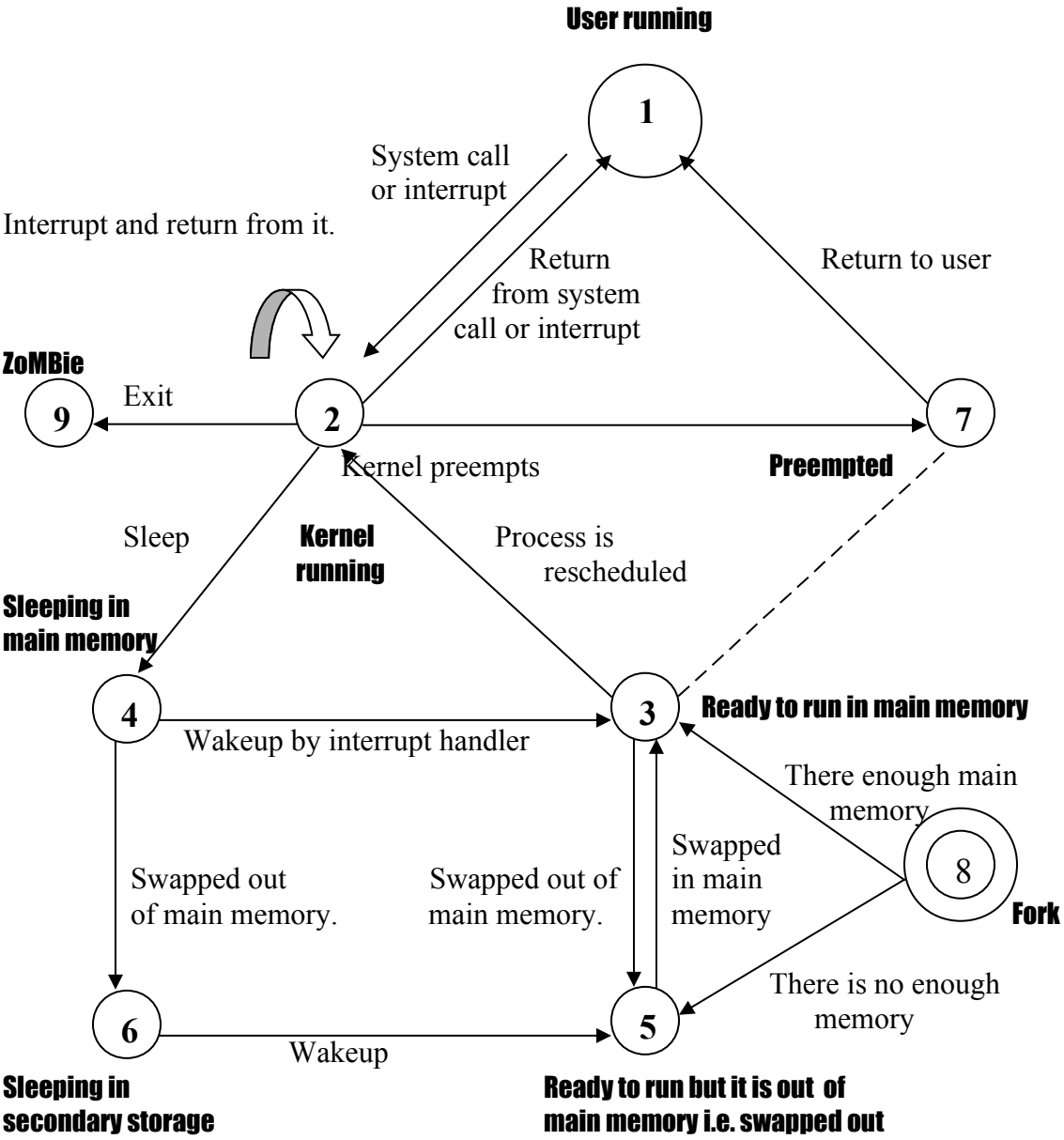
Here we will discuss all states more deeply. It is not at all necessary that every process must experience all of these states. Especially "SWAPPING" & "PREEMPTING" related states are not necessary for every process. But it is for sure that a process can go through maximum of these (9) states and not more than that.

See the diagram on the next page. First when parent process executes fork() system call, the process gets created and thus now it is in "CREATED STATE"(8^{th}).

After this stage, if there is enough main memory (i.e. enough RAM left), process will enter in "READY TO RUN IN MAIN MEMORY" state (3^{rd}). But suppose if there are already many processes running in main memory, and if our newly created process does not have enough memory to run, then the swapper process (i.e. process 0) swaps this newly created process "OUT OF THE ROOM" so as to make room for "ALREADY RUNNING PROCESSES". So in such circumstances out newly created process will enter in "READY TO RUN BUT SWAPPED OUT FROM MAIN MEMOTY" state (5^{th}).

From now we will consider that there is enough room in main memory for our newly created process. So for now it is in "READY TO RUN IN MAIN MEMORY" state.

Now when time comes, process scheduler will schedule this newly created process and thus it will now enter in "KERNEL RUNNING (i.e. Kernel mode)" state (2^{nd}). Here part of fork() system call completes.



When process completes execution of any system call (here for example the complete fork() system call), it enters in "USER RUNNING" state(1st). So process is now in "USER MODE". As stated above, when process is in "USER RUNNING" state (i.e. in user mode) it may come across various system calls, so it can arrive again and again in "KERNEL RUNNING" state and after completion of system calls it will go back to "USER RUNNING" state.

When process is in "KERNEL RUNNING" state (i.e. in the kernel mode), many interrupts(most commonly "SYSTEM CLOCK" interrupts) may occur to which kernel deals and after completion of interrupts process will be still in "KERNEL RUNNING" state. A "DOUBLE HEADED" rounded arrow indicates this.

Sometimes it may happen that when an interrupt occurs and kernel finishes with the interrupt, due to "MULTIPROCESSING NATURE" of kernel, it may schedule another process to run. So our newly created process, which is now in "KERNEL RUNNING" state gets preempted by kernel and thus enter in "PREEMPTED STATE" state (7th). In reality "PREEMPTED STATE" is similar to "READY TO RUN IN MAIN MEMORY" state (hence the dotted line is shown between these two states) but they are shown separately to stress that "KERNEL ONLY PREEMPTS THAT PROCESS WHICH IS READY TO RETURN TO "**USER RUNNING**" STATE."

Here as we said above that "PREEMPTED" and "READY TO RUN IN MEMORY" states are same(so now consider 7 as 3), kernel can swap out this process from the memory to make room for

other processes to run. So process may go to state of "READY TO RUN BUT SWAPPED OUT FROM MAIN MEMORY" state. Later the process scheduler will pull it back into main memory, where it will again enter into "READY TO RUN IN MEMORY" and then to "KERNEL RUNNING" and finally to "USER RUNNING" state.

As explained before when process is in "USER RUNNING" state and executes a system call it enter into "KERNEL RUNNING" state. While the process is in "KERNEL RUNNING" state and the system call is concerned with file I/O which needs some time for "DISK OPERATION" to get completed, our process (which is currently in "KERNEL RUNNING" state) goes to sleep and thus enters in "SLEEPING IN MAIN MEMORY" state (4^{th}). When disk operation by device driver completes, the hardware interrupts the CPU and interrupts handler awakes this slept process and thus process enters in "READY TO RUN IN MAIN MEMORY" state (As it is sleeping in main memory already). The horizontal arrow from (4) to (3) indicates this.

While the process is "SLEEPING IN MEMORY" state, it may happen that a totally new process may require memory to run and due to crowding of processes already in main memory, there may not be enough memory. So swapper process looks that one process is un-necessarily occupying the memory and it is in "SLEEPING" state. Thus swapper process pulls this slept process(keeping it in slept state) out of memory and thus our "SLEEPING IN MAIN MEMORY" state having process, will now enter in "SLEEPING IN SECONDARY STORAGE" state(6^{th}). Means temporary swapper process keeps it in a secondary storage.

Note that: - This "SLEPT AND SWAPPED" process when wakes up, cannot directly enter into "READY TO RUN IN MAIN MEMORY" state. First it goes to "READY TO RUN BUT SWAPPED" state and then swapper process pulls it back into main memory and then it enter in "READY TO RUN IN MAIN MEMORY" state.

When exit() system call is executed by the process obviously and ultimately it will be in "KERNEL RUNNING" state, from which it will go to zoMBie state(9^{th}) which is the final state of the process.

After all above discussion, concluding we can say that processes do not dictate the transition state. Rather dictation is mainly by kernel. But processes have some freedom to control transition states at user level…

- A process can create another process by fork() and thus can put its child in(8^{th}) state. But after that "STATES" all dictated by kernel.
- A process can make its own and any system calls to enter from "USER RUNNING" to "KERNEL RUNNING" state. But after that whole control goes to kernel and thus it is unpredictable when process will return to "USER RUNNING" state. It may, if kernel exits it forcefully or may take long time to return.
- A process can call exit() system call on its own and thus can enter in zoMBie state voluntarily. But this does not means that exiting is under control of process. The kernel without having exit() call by process, can explicitly force a process to exit and thus forcefully pushes into zoMBie state.

So ultimately we can say that, it is the kernel who mainly controls process states. The kernel has main(2) data structures to deal with "PROCESS STATES"…..

(1) The process table.
(2) The U area.

*** Contents Of Process Table :-** It is global.

1) The "STATE" field indicating which state the process is now in.
2) Fields that allow kernel to access process's U area and to locate process in the memory(for "CONTEXT SWITCH").
3) Field which gives size of the process.(So that kernel knows how mush memory space to allocate for this process).
4) Several user identifiers i.e. UIDs.(These are mainly necessary to determine process privileges)

5) Several process identifiers i.e. PIDs(These specify the relationship between different processes). There are created when process is created by fork().
6) Event descriptor,(Mainly require to wake up the process when it is in sleeping states).
7) Scheduling parameters.(These allow kernel to move the process between "USER RUNNING" and "KERNEL RUNNING" states).
8) Signal field.(This gives signals those sent to the process but yet not handled).
9) Various timers.(These are required to calculate "PROCESS SCHEDULING PRIORITY". Such as process execution time, kernel resource utilization time, user set time, etc).
10) Pointer to the per process region table also called as pregion entry.

* **Content of process's U area** :- This is local(i.e. private) to every process. Some of the contents are already seen
We will again see them all....

1) A pointer to the process table slot of currently running process.
2) Error code of current system call if any.
3) Return value of current system call if any.
4) Parameters of current system call if any.
5) File descriptor of all those files opened by this process.

We have already read about (1) (2) (3) (4) (5) previously.

6) Internal I/O parameters, which describe

- Amount of data transfer.
- Address of process's local buffer.
- File offset.
- Flags.
- I/O mode.

We have already read about (6)

7) Current directory and current root(for process's execution environment).
8) Process size limit and file size limit.
9) Real and effective UIDs to determine process's privileges.
10) Timer field, which contains information about "HOW MUCH TIME PROCESS SPENT EXECUTING IN "USER MODE" & "KERNEL MODE". This field also gives timers of child processes of this process.
11) An array, which indicates how, the process wants to react to signals.
12) "CONTROL TERMINAL FIELD", which identifies "LOGIN TERMINALS" of running process (if login terminal is there for this running process).
13) A "PERMISSION MODE FIELD", which masks mode settings on files that this process creates.

* **Layout Of System Memory** :- As explained in chapter(2), a program(more precisely a process) when loaded in memory, has 3 parts Text, Data and Stack. The "TEXT" section consists of binary-formed addresses of instructions (i.e. textual part of compiled code – like statements, equations etc). The "DATA" section consists of addresses of global data variables (if any). The "STACK" section consists of addresses of local data variables functions etc.

We also explained that when compiler compiles source code of a program, it creates above addresses for "VIRTUAL MACHINE". Means these addresses are machine independent and thus irrespective of machine's actual physical memory. So compiler generated addresses are for "VIRTUAL ADDRESS SPACE" having some finite range 0 to some compiler limited value.

The physical memory is also finite ranged, from 0 to the max byte offset equal to the amount of memory on the machine, minus 1(because we count from 0).

Now it is the duty of operating system (OS) to convert "COMPILER GIVEN" virtual addresses into actual physical memory addresses. The part of the OS i.e. the part of the kernel, which does this translation of "VIRTUAL ADDRESSES" to "ACTUAL PHYSICAL ADDRESSES", is called as "MEMORY MANAGEMENT SUBSYSTEM".

Here two things must be kept in mind….

(1) Several instances of one program may exist at a time. As compiler generated virtual addresses are same for all these instances, kernel has to map every "INSTANCE ADDRESS" at different physical location in RAM so that all instances will run independently.

(2) Though several instances of same program co-exists, it does not mean that Text, Data & stack parts of all these instances are separate. In fact Text and Data(i.e. global) have unique place for all above instances but local data(i.e. stack) and local functions are separate for every instance.

Regions: - UNIX SYSTEM V divides "COMPILER GIVEN VIRTUAL ADDRESS SPACE" into logical spaces known as regions. Regions is a contiguous area of a process, so if a process has Text, Data and Stack parts, there will be "TEXT REGION" for Text part, "DATA REGION" for data part and "STACK REGION" for stack part of that process.

But if several other processes execute one common process, then all these processes will share "ONE TEXT REGION" of common process. Means there is no need of duplicating "TEXT REGION" of common process into memory of other processes. This is called as sharing of a region.

In chapter (2), we explained some data structures of a process like U area, per process region table, process table and region table. We will consider these data structure in more details…

The U area: - As explained in chapter (2), when the system is installed, its source code is compiled and loader creates a variable called as "U" which is the name of "U AREA". Compiler also gives a fixed virtual address to it. This address is global and thus known to other parts of kernel too. This is particularly important because the code of kernel, which actually does the context, switch, require this value continuously.

As usual, kernel also does mapping of this virtual address of "U AREA" to a physical memory location and remembers this physical memory location too. Kernel also known the part of memory management system, which do this translation of "U AREA's VIRTUAL ADDRESS" to "U AREA's PHYSICAL ADDRESS". As kernel knows this part of memory management system, it can dynamically tell this part to map "U AREA's VIRTUAL ADDRESS" to some other physical memory location.

Now note that, every process when gets executed has its own private "U AREA". It also has some "VIRTUAL ADDRESS" and corresponding mapped "PHYSICAL ADDRESS" too.

So when this process enter in kernel mode, kernel maps its global virtual address of "U AREA" to physical memory address of process's local "U AREA", by telling the previously mentioned part of memory management system.

Thus if process A is running in kernel mode now and its local "U AREA's PHYSICAL ADDRESS" is say UA. Suppose the virtual address of kernel's global U area is UK, then kernel associates UK to UA and thus can access process's U area of process A.

When process returns to user mode, kernel again resets its UK to its previous global value. If it now schedule process B having its U area's physical address as UB and when it enters in kernel mode, now kernel's UK points to UB.

- Hence it is said that, though two processes have their U areas at two physical memory address, kernel access them via its single virtual address of its own U area.
- Also remember that a process can access its U area only when it is in kernel mode (this indirectly tells us that process accesses its U area by taking help of kernel) and never in user mode.

- It also must be kept in mind that, kernel at a time can access U area of one process only. E.g. at a time UK can point to either UA or UB depending which one of A or B is currently running in kernel mode.
  The contents of "LOCAL U AREA OF A PROCESS' are already explained. Mainly it contains pointer to "PROCESS TABLE ENTRY" of this process.

*** <u>Per process region table</u>: -** This contains pointer to the "REGION TABLE" entry of this process. This data structure is local to every process. As it contains pointer to region table entry of this process, it has addresses (virtual) of Text, Data and Stack parts of this process. "PER PROCESS REGION TABLE" is also called as pregion for the sake of simplicity.

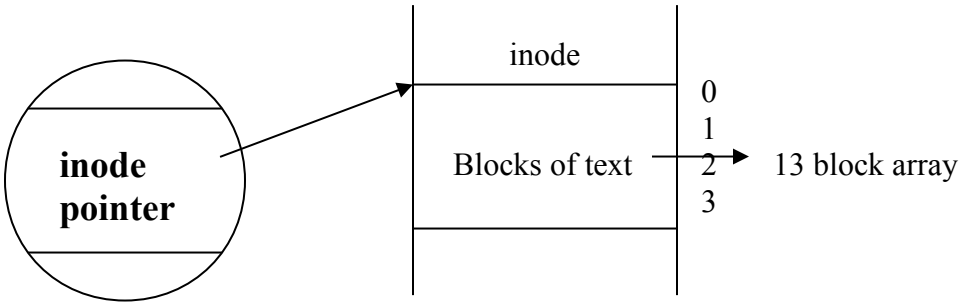
*** <u>Process table</u>: -** This is global, process table contains pointer to "PER PROCESS REGION TABLE" entry of this process. This entry is called as pregion entry. This entry contains 2 things…

(1) Pointer of "REGION TABLE" entry, which in turn contains starting virtual addresses of Text, Data and Stack regions of this process.
(2) A permission field, which indicates the type of access given to the process. Thus it has one of the three values "READ ONLY", "READ - WRITE" and "READ - EXECUTE".

- **Note: -** Though we said that "PREGION ENTRY" is located in process table, some other UNIX implementations may keep it in "U AREA".

*** <u>Region table</u>: -** This is global; it mainly has entry for the each active region (of all processes. Not only of this process). **All of its content are….**

- Pointer to the inode of the file of process (**Note: -** Though process is a dynamic state, it has a program which in turn is located in a file say temp.0), which is now loaded into the region.
  This statement may confuse us that Text, Data and Stack regions of a process point to same location (i.e. inode). It is partially correct. Means, yes they are pointing to same inode of executable file, but their pointing location inside that inode is different. Means "INODE POINTER" field in "TEXT REGION" will point to that inode block (starting block) where text blocks start. Similarly "INODE POINTER" field of "DATA REGION" will point to that inode block where data starts.



TEXT REGION

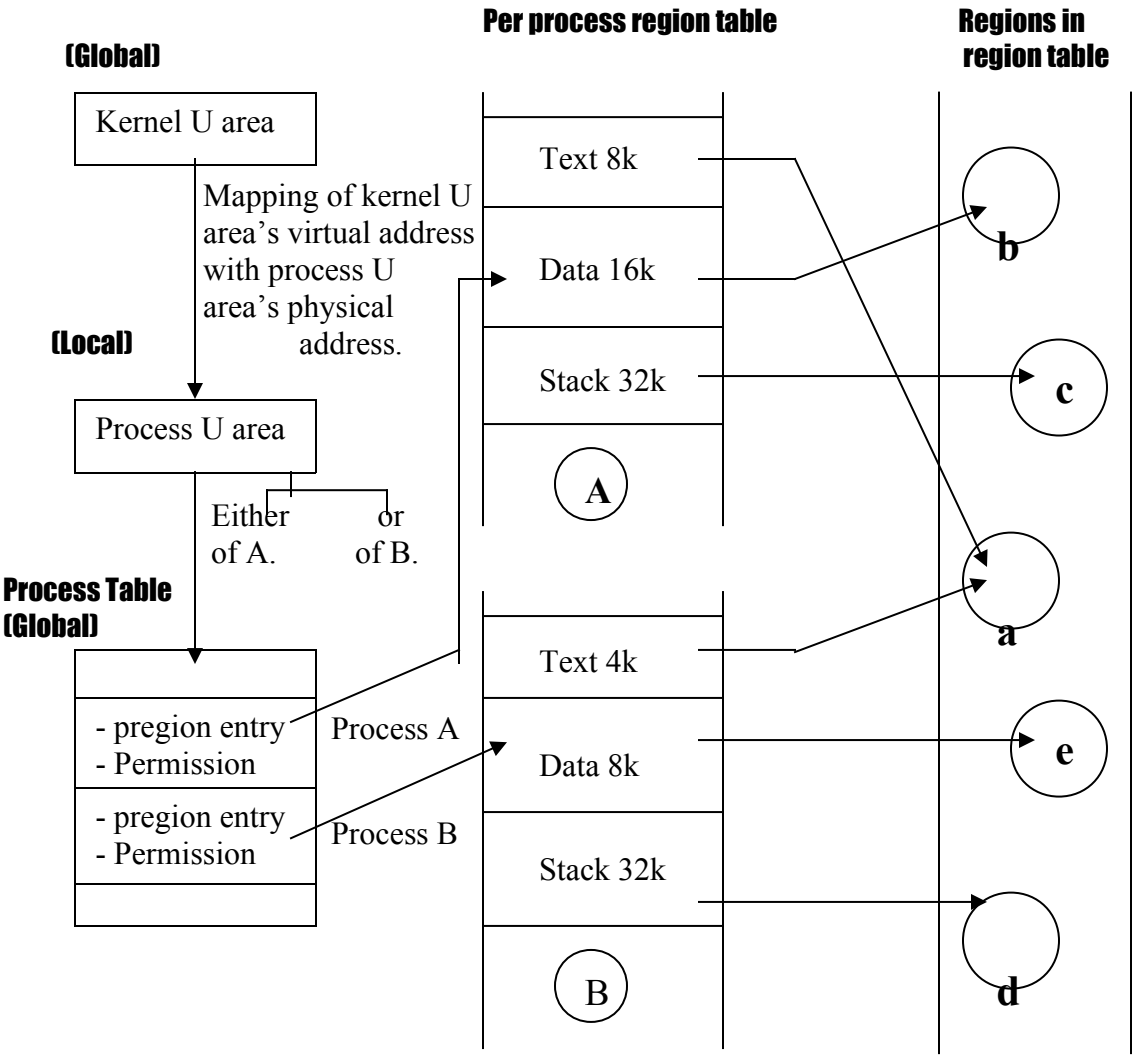
- Type of the region (i.e. Text, Data or Stack or some shared memory).
- Size of the region.
- Location of the region actually in physical memory. (Page tables)
- Status of region, which is either "LOCKED", "IN DEMAND", "BEING LOADED INTO MEMORY", "ACTUALLY LOADED INTO MEMORY (i.e. valid)".
- Reference count, which indicates how many processes currently, accessed this region.
  Above Study of 4 data structures of a process tells us a very important analogy.

* Process table ***analogues*** to user file descriptor table (Because just like file descriptor, it has pregion entry).

* Per process region table ***analogues*** to File Table (Because as file table has "MANY YO ONE" relationship with inode table due to file sharing, per process region table also has "MANY TO ONE" relationship with region table due to region sharing)

* Region table ***analogues*** to inode table. (Because as inode table has sharable region and as inode table has "REFERENCE COUNT ", region table also has a "REFERENCE COUNT") Striking differences between "FILE INODE" – "PROCESS REGION" analogy are...

- File table is global while per process region table is local.
- File manipulations only use local U area while process manipulations requires both kernel's global U area and process's local U area.
- User file descriptor table is local while its analogues process table is global..



As you can see in the figure, there are 2 processes, A and B. They have common Text region "a"(hence shared), but have different "DATA" and "STACK" regions,(hence private - local). Though shared region "a" is one, processes access them via 8k and 4k virtual addresses. Note that though virtual addresses 8k and 4k are different, they both point to the same physical memory location of "a".

* **<u>Pages and page table</u>: -** This is the default memory format of UNIX SYSTEM V used in next chapters. But different implementation of UNIX may use other formats.

The memory management hardware divides physical memory into equal sized blocks called as pages. So memory management architecture based on pages. This same idea by Microsoft's operating systems too.

The page size may vary from 512 bytes to 4K, and it is hardware dependent. Means memory management hardware and memory chip hardware decides this size of page.

Obviously every memory location (i.e. physical memory address) has to lie inside one of the page. Thus every addressable memory location can be addressed by "PAGE NUMBER & BYTE OFFSET INSIDE THAT PAGE" combination.

E.g.: - Suppose we want to access physical memory location 5555 and if we assume hardware dependent page size is 1k each, then 5555^{th} memory location will be in 5^{th} page (counted from 0) at 434^{th} location (counted from 0).

- 0^{th} page => 0 to 1023
- 1^{st} page => 1024 to 2047
- 2^{nd} page => 2048 to 3071
- 3^{rd} page => 3072 to 4095
- 4^{th} page => 4096 to 5119
- 5^{th} page => 5120 to 6143

Now in 5^{th} page byte offsets start from 0.....

5^{th} page 5120 to 6143

0^{th} byte offset => 5120

1^{st} byte offset => 5121

.
.
.
.

434^{th} byte offset => 5555

.
.
.

1023^{rd} byte offset => 6143

* So the(page number, byte offset) combination will be(5, 434).

If a machine has 2^{32} byte of physical memory and the page size is 1k, then there will be 2^{22} pages of memory.

1024 byte = 1 page

2^{32} byte =?

$$\frac{2^{32}}{1024} = \frac{2^{32}}{2^{10}} = 2^{22}$$

* In other words the last page number will be 2^{22}. So a 32 bit address can be treated as 22 bit page number and 10 bit byte offset.

E.g. :- Suppose on address in here is 58432, then its binary equivalent is .....

1011000010000110010 => 19 digits to make it of 32 bit pad OS to left.

0000000000000101100001 0000110010 => 32 bit

22 digit → In hex 161^{st} page, In Dec 353^{rd} page

10 digit → 32^{nd} byte offset, 50^{th} offset

Kernel (i.e. OS) by using memory management policies can assign physical pages of memory to a region. **Note: -** region is logical and contiguous but it is not necessary that pages should be contiguous or in a particular order.

OS does dividing physical memory into physical pages for greater flexibility for memory management. This is just like system divided default block size of 1024 and divide disk into multiple blocks each of 1024 byte. So pages are quite analogous to blocks. Dividing disk into blocks reduces disk fragmentation, so analogously dividing physical memory into pages reduces memory fragmentation.

Similar to this, region table is analogous to inode table, so region is analogous to an inode.

Thus as inode contains file blocks (in array of 13 members) region contains pages. But note that file blocks in inode are logical and mapped to actual physical blocks. Similarly region does not contain actual physical pages but contain logical pages which are then mapped to actual physical pages.(Just like 0^{th} block in disk block array in inode points to some actual physical data block).

Logical page number.	**Physical page number.**
0	177
1	54
2	209
3	17

As region is continuous, logical page number are also sequential started from 0. If we see above mapping of logical page number to physical page number, we can say that logical page number is an index into physical page number array. In other way we can write int ppn[] = {117, 54, 209, 17}; As this is an array we can call it as page table.

Now extending our knowledge of region, region table, page and page table, we can say that....

Region table contains regions of a process but region contains pages means region contains page table.

Thus region table contains page table. As array has a starting address, we also can say that region table contains pointer to page table.

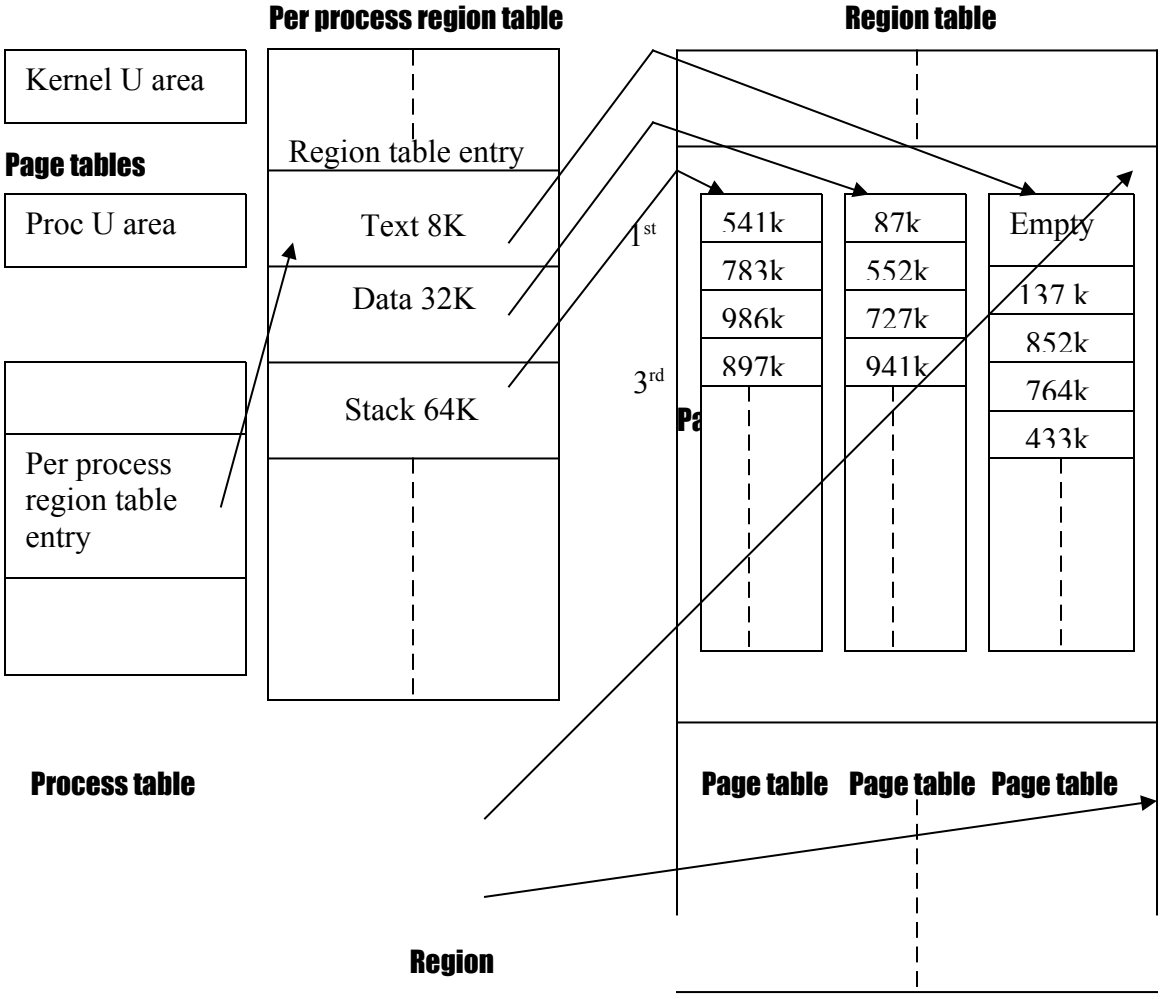
Again come back to our disk block – memory page analogy. Though inode of a file contains disk blocks, the process must have permission to read or write the disk blocks. Permission field in inode verifies this permission.

Similar to this memory location also has permission to read or write on it. This permission flag is also stored in page table entry along with the page number which checks whether reading or writing on respective page is allowed or not.

E.g. :- Suppose there is a running process. Also suppose that page size is 1K, and the process wants to access a virtual memory address 68432. We want to calculate the OS mapped physical memory location corresponding to above virtual memory address.

See the figure on this page. Note: - Number like 8K, 541K are not sizes but are starting addresses of that area in the figure.

Now “PER PROCESS REGION TABLE”(i.e. pregion entry) shows that this process’s text section is from 8^{th} to 31^{st} K, Data section is from 32^{nd} and 63^{rd} K and stack section is from 64^{th} to X^{th} K.



So our desired virtual memory address 68432 i.e. 68432/1024 = around 67 K, lies in stack section.

The stack section points to a page whose starting address is 541k. This page is in a page table whose other pages start from 783rd, 986th 897th K addresses and so on. Forget about these number right now. [Again note that, physical pages need not to be continuous. Hence written randomly.]

As our required address 68432 is in stack region, according to stack its pointer moves as elements are pushed and popped. Here we assume that stack will grow towards higher address.

Our required virtual address is 68432 and stack's starting address is 64K, means 65536. Subtracting them we get 2896. Means required address is at byte offset 2896 from starting address 65536.

Now see that 65536 points to 0th page of page table. If each page is assumed of 1K, 0th page will have virtual addresses 0 to 1023, 1st page will have 1024 to 2047 and 2nd page will have 2048 to 3071 and so on.

Our required byte offset is 2896, which is obviously in 2nd page of this page table. This page starts from 2048, so 2896 is 848 away from 2048.(i.e. 2896 – 2048 = 848).

Thus the (page number, byte offset in page) combination will be (2, 848). Means our required virtual memory address 68432 is in 2nd page (counted from 0) at 848th byte offset. From figure we also know that this 2nd page is located at 986K physical memory addresses.

*** <u>Memory Management Hardware</u>: -** As discussed up till now, memory management largely does the task of translation of logical virtual memory address to actual physical memory address.

To do this task, machines need memory management hardware, which is used by kernel (i.e. OS) to do memory management tasks.

- The hardware used for this purpose is registers and cache. When a process executes kernel informs this hardware about the virtual addresses and memory management loads respective registers.
- The memory management system uses 3 registers called as register triple. First register contains address of a page table in memory. Second register contains first virtual address mapped by triple and third register contains control information, such as number of pages in this page table and also page access permission (i.e. “READ ONLY”, “READ - WRITE”).
- So when kernel executes a process, it looks for pregion entry and loads respective register triples with corresponding to pregion table entry.

Here (2) things must be kept in mind…

1) If a process tries to access memory location, which is outside of its virtual address space, the hardware causes an exception and OS handles it accordingly. Means from figure on Page 133, if Text region size is of 16KB (means from 8K to 24K) and process accuses virtual address at 26K, hardware will give exception.
2) Similarly if a process tries to access a memory location without permission, means a Text region is write protected and process tries to write at somewhere in this Text region, then too hardware will cause exception. It should be noted that in both above cases, process could exit normally.

*** <u>Memory Layout Of Kernel</u>: -** As kernel itself is a set of programs, it also has its own memory layout. Thus it also has its own Text, Data and Stack regions. Obviously it also has its page tables and register triples too.

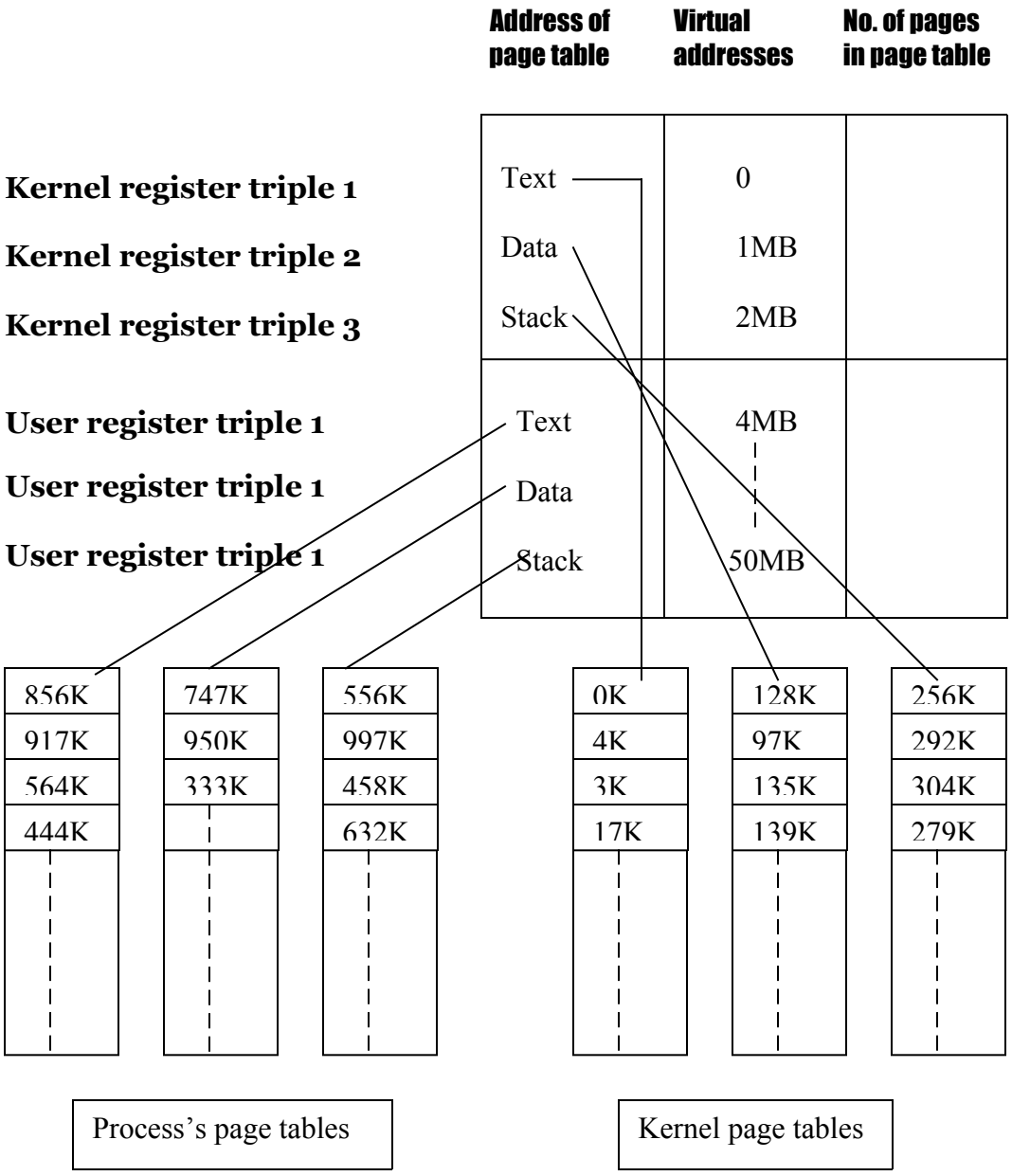
As seen in 2nd chapter, kernel’s code (text part) and data sections are global and sharable to all processes. Obviously kernel’s Text and Data regions must be kept permanently in the memory and must be kept at fixed locations so that process could found them at certain location.

Thus when system boots, it loads kernel’s text section in memory and set up respective tables and registers to map kernels own virtual addresses to actual physical memory addresses.

Now when a process executes in “KERNEL MODE” kernel allows that process to access kernel’s own Text and Data sections. But when process executes in “USER MODE”, kernel prohibits that process from accessing kernel’s Text and Data sections.

Some machines give kernel special registers for its own Text and Data regions. So that when a process is in “KERNEL MODE”, system allows that process to access kernel’s Text and Data region via these special registers. But when the process is in “USER MODE”, system prohibits this process to access these special registers.

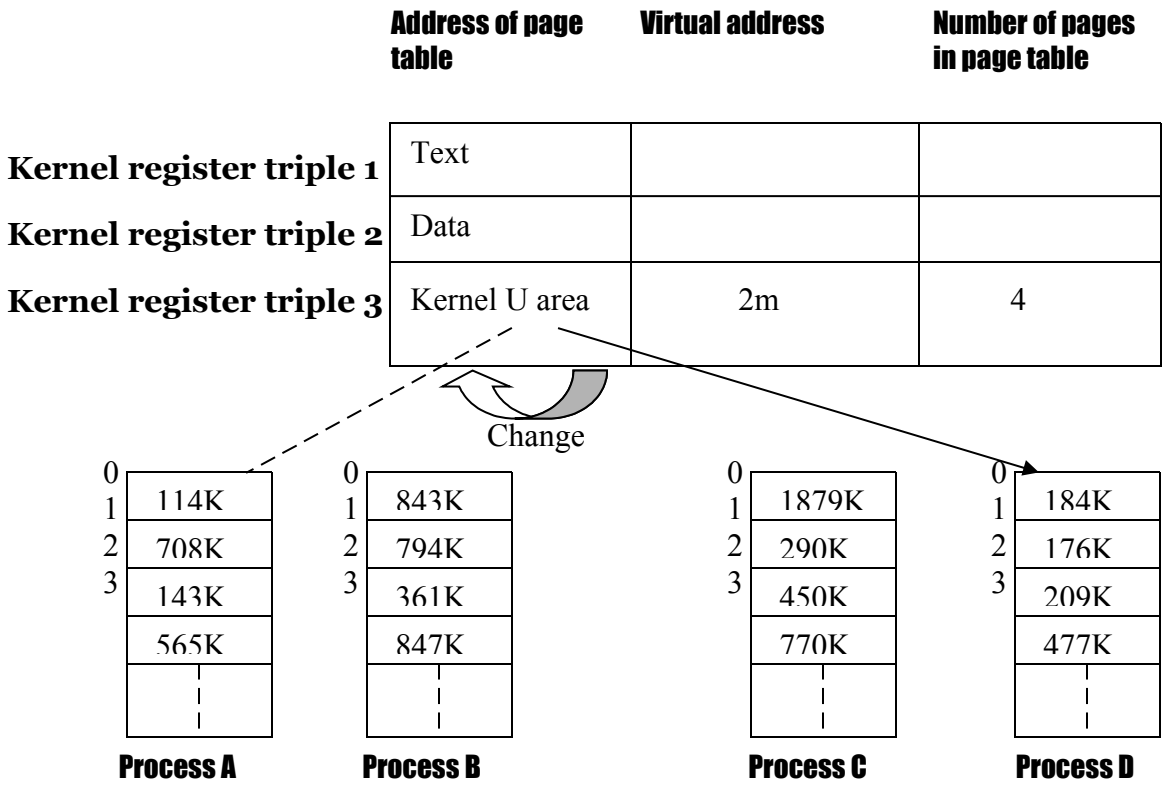
So “KERNEL REGISTER TRIPLE” works in kernel mode and “USER REGISTER TRIPLE” work in user mode of a process.



This figure shows virtual addresses of kernel from 0 to 4 MB – 1 and virtual addresses of user process from 4 M onwards. Kernel register triples are pointing to kernel's page tables while user register triples are pointing to process's page tables. When process is in "KERNEL MODE", it can access kernel page tables via kernel register triples and in user mode, using kernel register triple is denied.

In some machines, kernel's virtual addresses are made identical to the actual physical memory addresses, so no special mapping is required. But of course mapping of kernel's U area virtual address to the process's U area's physical memory address is must.

*** How Mapping Of Kernel U Area's Virtual Address To Process's U Area's Physical Memory Address Is Done?**



Suppose that kernel's Text, Data regions are referred by Kernel Register Triple 1 and 2 respectively. The Kernel Register Triple 3 is referencing to kernel's U area, which is currently pointing to U area of process D, which has 4 pages in its page table. The U area of process D is at 2m virtual address.

Now when kernel wants switch to process A it just copies respective process's page table information into the 3rd register row and changes 1st column reference such that, now it is pointing to 0th page of process D (physical address 184 K) which will change to 0th page of process A (physical address 114K).

Note: -

- Entries for register triple of kernel, number 1 and 2 will never changes for the kernel.(to allow sharing)
- Third register will always point to U area of a process, but only of currently running process. The entries in 3rd register row 2M for virtual address of process D and 4 for number of pages in page table of process D, will now change to something 3M(which may be the virtual address of process A) and 5(which may be the number of pages in page table of process A).

* <u>Context Of A Process</u>: - Context of the process consists of....

(a) Process's address space.
(b) Contents of hardware register (mainly CPU).
(c) Process related kernel's data structures.

Author Maurice. J. Bach of "The Design of the Unix operating system" calls above 3 contents as "USER LEVEL CONTEXT", "REGISTER CONTEXT" and "SYSTEN LEVEL CONTEXT" respectively.

(a) <u>Process's Address Space Or User Level Context</u>: - This consists of...

- Process's Text, Data, Stack and if present the "SHARED MEMORY". In other words these all things together can be termed as "PROCESS's VIRTUAL ADDRESS SPACE".

- Note that due to swapping or paging some or all-above contents may not reside in main memory and hence they may be in "SECONDARY STORAGE DEVICE"(See "STATES OF PROCESS at the beginning of this chapter"). Then too they are considered to be a part of this user level context.

(b) <u>Contexts Of Hardware Registers Or Register Context</u>: - This consists of….

- Address of the next instruction which is going to be executed.(Recall about fetch execution cycle). The program counter specifies this address. This is a virtual address either located in kernel's memory or located in user's memory.
- The Process's Status Register (PS) specifies the process related hardware status. This thing is specific for this running process only. Hence the phrase "PROCESS RELATED" is used.
  This PS contains sub fields, which indicate…
  - ⇨ Some sub fields indicate whether the result of a recent process's calculation is 0 or negative or positive.
  - ⇨ Some sub fields indicate whether register is overflowed.
  - ⇨ Some sub fields indicate whether a "CARRY BIT" is set.
  - ⇨ Some sub fields indicate current processor execution level for interrupts.
  - ⇨ Some sub fields indicate recent execution mode of the process, i.e. whether kernel mode or user mode.
  - ⇨ Some sub fields indicate current execution mode, which is necessary to determine whether the process can execute privileged instructions and whether the process can access kernel's virtual address space or not.
- The stack pointer (SP) contains current address of the next entry in the kernel's stack or in the user's stack. Obviously this is determined by the current execution mode. Means "THE ADDRESS OF NEXT ENTRY" will be of kernel stack if the current execution mode is "KERNEL MODE" AND similar to this "THE ADDRESS OF NEXT ENTRY" will be of user's stack if the current execution mode is "USER MODE".
  Though here it seems that OS is dictating the hardware, there are 2 things, which are dictated by the machine's architecture and not by the OS. These things are….
  - ⇨ Whether Sp should point to "NEXT FREE ENTRY IN THE STACK" or "LAST USED ENTRY IN THE STACK".
  - ⇨ Direction of growth of stacks i.e. whether stack grows to higher addresses or to lower addresses.
- General-purpose registers (AX, BX, CX, DX) contain data generated by the process during its execution.

c) <u>Process Related Kernel's Data Structures Or System Level Context</u>: - This Consists of…

I. The process table entry. As global, it is accessible to kernel.
II. The U area of the process (local to process).
III. pregion entries, region tables and page tables. If regions are shared then those regions become part of content of every process, because though "SHARED REGIONS" are common to those all processes, every process executes independently.
IV. Which part of the process's address space is not residing in memory (if any). Means which part of the process's address space is swapped out from the ram and hence residing in the "SECONDARY STORAGE". Determining such part of the process is the task of "MEMORY MANAGEMENT UNIT".
V. The kernel stack contains stack frames of kernel's procedures done during the process in "KERNEL MODE". Note that though all processes have "ONE SINGLE COMMON KERNEL", they all have "PRIVATE COPY" of their kernel stack. Thus each process's kernel stack is specific for that specific process's kernel mode tasks. Usually "KERNEL STACK" is kept in U

area of respective process. But this is not a compulsion. Some UNIX flavors may keep it at some another, different and independent memory location. The basic idea is that the kernel after returning from context switch must point to respective process's kernel stack and also must be able to point its "STACK POINTER" at correct position of kernel stack. The (4) things seen above (except 5th) are together called as "STATIC PART" of system level context. **Note** that a process always has one and only one "STATIC PART" of system level context throughout its lifetime.

VI. There is yet another stack of layers where each layer is made up of system level context (i.e. combination of above 5 contents). Each layer contains the information about recovery of previous layer. Some implementations also include "REGISTER CONTEXT" in this layer too.
5th and 6th part of system level context is called as "DYNAMIC PART" of system level context.
As mentioned above, though a process can have only one "STATIC PART", it may have variable number of "DYNAMIC PARTS". This "DYNAMIC PART" is viewed as a stack of "CONTEXT LAYERS" which get pushed and popped according to various events.

*** <u>Process Of Context Switch</u> :-**

- Kernel pushes a context layer when an interrupt occurs or when process makes a system call or when process makes context switch.
- Kernel pops a context layer when kernel return from handling the interrupt or when process return to "USER MODE" after completing the execution of system call or when a process does context switch again.
- Kernel always pushes the context layer of old process and pops the context layer of new process.
- All these things are possible because process's "PROCESS TABLE" stores the necessary information to recover the current context layer.
- A process runs within its current context. In other words we can say that, process runs within its current context layer.
  The number of context layers depends upon....
  (a) How many interrupts levels are supported by machine.
  (b) One layer for system calls.
  (c) One layer for "User level context".
  So if machine supports 5 levels of interrupts i.e. software interrupts, disk interrupts, terminal's interrupts, other peripheral interrupts and clock interrupts, then the number of "CONTEXT LAYER" will be 7 (i.e. 1 for each interrupt level hence 5 for five interrupt levels + one for system calls + one for "USER LEVEL CONTEXT").

* Saving Context Of A Process: - As **seen above when...**

- ⇨ Interrupt is received or
- ⇨ Process executes a system call or
- ⇨ Kernel does a context switch; the kernel pushes a new system context layer into the stack. This, in another term, is called as "SAVING CONTEXT OF A PROCESS".

So we have to study following 3 situations in greater details....
(1) Interrupts (also exceptions) occurrence.
(2) System call making.
(3) Kernel's context switch.

(1) <u>Interrupts And Exceptions</u>: - Any OS is responsible for handling the interrupts. Interrupts are either hardware interrupts or software interrupts. OS is also responsible for handling exceptions.(like page faults)

As seen in chapter 2 if CPU 's processor execution level is set to an interrupt level and if currently CPU is execution an instruction having interrupt of higher level than the previously set level,

then CPU accepts this newer higher leveled interrupts before executing next instruction. After accepting the interrupts now CPU sets its "PROCESSOR EXECUTION LEVEL" to the level of current interrupts, so that no other interrupts of equal or lower level to that of current level can occur while it is dealing with the current interrupt. This strategy preserves kernel data structure integrity. The handling of interrupts occurs in following sequence...
First we will see the algorithm of interrupts handling...

```
01: Algorithm: int hand
02: Input: nothing
03: {
04:    save(i.e. push) current context layer;
05:    determine source of the interrupt;
06:    find "interrupt vector";
07:    call "the interrupt handler";
08:    restore(i.e. pop) the previously saved context layer;
09: }
10: output: nothing
```

Sequence is as follows...

(1) Kernel saves current "REGISTER CONTEXT" of the executing process and pushes a new context layer in the kernel's stack.

(2) Then it determines (or say finds out) the "SOURCE" or the "CAUSE" of interrupt. Then it identifies the type of interrupts and the unit number of interrupt. This occurs as follows....

When kernel receives the interrupt, it gets a number (usually supplied by the process either directly or indirectly). This number is used as an "OFFSET NUMBER", by which kernel goes to "INTERRUPT VECTOR TABLE"(IVT) and then uses this "OFFSET NUMBER" to get the entry of the interrupt vector table. The contents of IVT differ from machine to machine (hence IVT is machine dependent). **Usually it contain**

- ⇨ Interrupt number.
- ⇨ Address of the interrupt handler function (or routine) for that specific interrupts.
- ⇨ A way of finding parameters for that interrupt handler function.

Note: - If interrupt handler functions are already written in specific order, then there is no need of "INTERRUPT NUMBER's SPECIAL ENTRY" into the IVT. Line number 0 to n of entries, themselves can be considered as "INTERRUPT NUMBER" or say index into IVT

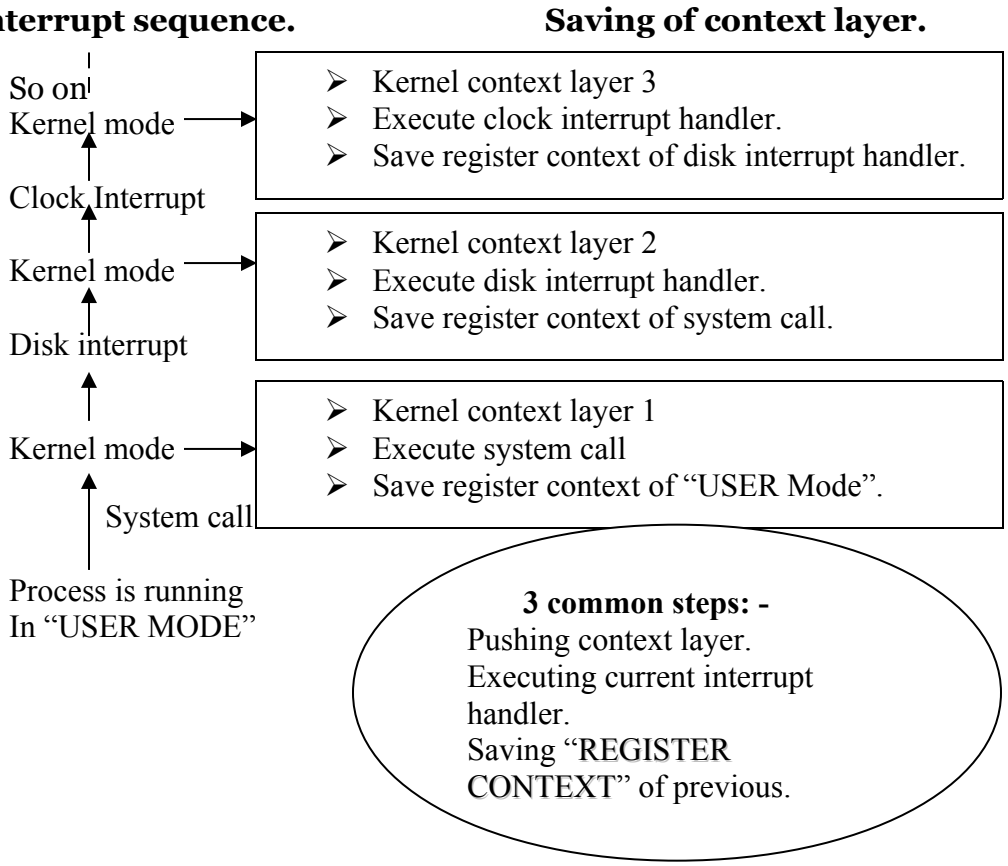
A sample of IVT may be like following: -

Interrupt No.	Names of interrupt handler f^n.	
0	clockintr()	For clock interrupt.
1	diskintr()	For disk interrupt.
2	ttyintr()	For terminal interrupt.
3	devintr()	For device interrupt.
4	softintr()	For software interrupt.
5	otherintr()	For any other interrupt.

So for example, if a terminal interrupts a system then kernel gets the "INTERRUPT NUMBER" '2' from the hardware, it goes to IVT, goes to the number 2 entry and finds out interrupt handler ttyintr().

(3) Now kernel knows the interrupt handler ttyintr(). So it calls this function and completes the task of desired function of the interrupt.

(4) Then kernel executes "SPECIFIC MACHINE INSTRUCTIONS" to restore (i.e. pop) the previously saved(in step 1) "REGISTER CONTEXT" and "KERNEL STACK" having context layer and then starts execution of this context layer(as if nothing is happened).



The figure on last page gives one example of a running process. Here process first executes in "USER MODE" and while in user mode it executes a system call. As a consequence of system call it enters in "KERNEL MODE". Here kernel saves "REGISTER CONTEXT" of user mode, pushes "KERNEL CONTEXT LAYER 1" into the stack and executes the system call.

Now suppose there occurs a disk interrupt, kernel saves "REGISTER CONTEXT" of system call, pushes "KERNEL CONTEXT LAYER 2" and then executes the disk interrupt handler function.

Then if a clock interrupt occurs, kernel saves "REGISTER CONTEXT" of disk interrupt handler, pushes "KERNEL CONTEXT LAYER 3" and executes clock interrupt handler function.

(2) <u>Making Of System Call</u>: - Up till now we were considering system calls as like other functions. Externally on user level it is quite correct too. But internally, to kernel, system call is a special kind routine allowing kernel to pay additional attention.

All system calls are declared in a 'C' library. So when a process makes a system call, linker links function call to the "NAME Of IT" in that 'C' library. The 'C' library of system calls is generated in such a way that when a system call is made, library invokes an "SYSTEM CALL SPECIFIC" instruction called as "OPERATING SYSTEM TRAP".

<u>Note: -</u> Library itself does not have code of system call. It just has its name and according "trap" instruction. The actual code of system call resides in the kernel.

Note that still we are in "USER MODE". Now after creating "OPERATING SYTEM TRAP", library functions pass a unique number (just like to interrupt handler) to the kernel. Passing of this unique number takes place by one of the 2 ways depending upon machine.

- ⇨ Either as a parameter to the instruction itself.
- ⇨ Or on the stack.

Now we enter in the kernel mode and kernel is ready to handle the "OPERATING SYSTEM TRAP INSTRUCTION". Similar to IVT, there is a "TABLE OF SYSTEM CALLS", in which kernel looks for above "SPECIAL NUMBER" as an index and when finds it gets the address of the required system call function. This address is obviously the "ENTRY POINT" of the system call.

Thus here we get the entry point of system call. But what about the parameters? We know that, process has 2 stacks "USER STACK" and "KERNEL STACK"(see chapter 2). In the frames of the user stack, there are parameters to the called function. So by using "USER STACK POINTER" it adds or subtracts (depending upon whether stack grows to higher address or to lower address) an offset to the "USER's STACK POINTER". This offset depends upon number of parameters to the system call. Hence now kernel got the parameters of system call. Then it copies these parameters from stack to the process's U area and starts executing the code of the system call by entering inside the entry point (previously found)

Recall that one of the content of U area is parameter of current function.

Note that if system call has more than one parameter, then which should be considered first? depends upon the system implementation or on compiler's calling convention.

After Executing The System Call's Code There Are 2 Situations

- ➢ Either success with or without return value.
- ➢ Or error with return code.

(a) <u>Success with or without return value</u> :- We know that "PROCESS STATUS" registers has a sub field with a "CARRY BIT". "CARRY BIT" is used in "ERROR SITUATIONS". Here, there are no errors; hence kernel clears this "CARRY BIT" and copies "RETURN VALUE (if any)" into the 0^{th} and 1^{st} register of the "REGISTER LEVEL CONTEXT". *0^{th} and 1^{st} register of "REGISTER CONTEXT" are called as "DATA REGISTERS".*

(b) <u>Error with return code(Usually -1 in Unix)</u> :- If kernel determines that execution of system call is completed with some error, then kernel sets "CARRY BIT" in "PROCESS STATUS" register and copies the error number into the 0^{th} register of the "REGISTER CONTEXT"**.** Now the job of system call is done means job of "OPERATING SYSTEM TRAP" is done. So kernel returns to the library from which "TRAP" was created. While coming back to library, library checks status of PS register and values in register 0 and 1 and then takes appropriate action. Usually library returns these values (i.e. return values or error code) of system call to the caller process and process further takes according actions.

The algorithm for above mentioned system call interface in context switch mechanics is as follows….

```
01 : Algorithm : syscall      /* from invocation of system call *
02 : Input : system call number(takes from "trap")
03 : {
04 :   find entry in "system call table" corresponding to system call number;
05 :   determine number of parameters to the system call;
06 :   copy parameters from "user address space" to U area;
07 :   save current context for abortive return;
08 :   invoke the code of system call reside in kernel;
09 :   if(error during execution of system call)
10 :          {
11 :            set register 0 in "register context" to the error number;
12 :              turn on "carry bit" in "process status" register in register context;
13 :          }
```

```
14 :            else /* success */
15 :              set register 0 and 1 in "register context" to return values of system call;
16 : }
17 :  output : result of system call.
```

Concluding remeMBer one point, that library contains many functions, but system calls are few. This clearly shows that many library functions have common entry point of system call function.

Then too, we use library functions more commonly than the actual system call. This is because library functions take care of many things, which can be left, ignored by the process. Also library can handle many versions of one system call. E.g.: - exec system call has library flavors like execl, execv, execle, spawnl, spawnlp, etc.

(3) The Actual Context Switch -: As seen in the topic of "PROCESS STATE TRANSITION", kernel allows "CONTEXT SWITCH" in following 4 situations: -

I. When a process puts itself into sleep.
II. When a process exits.
III. When a process returns from a system call and ready to go back in "USER MODE", but scheduler does not consider it as eligible process to run.
IV. When a process returns to "USER MODE" as kernel finishes handling an interrupt, but scheduler does not consider this process as eligible process to run. While making "CONTEXT SWITCH" kernel strongly maintains integrity of its data structures by…
    - ➔ Prohibiting arbitrary context switch.
    - ➔ All necessary updates of kernel's data structure are properly done, e.g. queues are linked properly.
    - ➔ Appropriate locks are set so as to prevent other processes to use same resource.
    - ➔ Ensuring that no un-necessary data structures are locked.

E.g.: - If kernel allocates a buffer and reads a block into it and process goes to sleep for waiting on I/O completion from disk, then kernel keeps the buffer locked.

On other hand, a process if executed link() system call, then kernel releases the lock of the first inode before locking the second inode to avoid race conditions.

Out of the 4 situations for "CONTEXT SWITCH" 3 are optional (means kernel may or may not do the context switch) but the 2nd one (i.e. b) is must. Means "CONTEXT SWITCH" is must when a process exits.

The Process Of Context Switch: - The procedure of context switch is similar to that of handling an interrupt or handling a system call (both are already discussed). The big difference is that, in interrupt and system call handling situations, kernel restores previous "CONTEXT LAYER" of the same process, while for handling of "CONTEXT SWITCH" kernel restores "CONTEXT LAYER" of different process.

But in "CONTEXT SWITCH" which process is to be scheduled (i.e. which process's context layer is to be restored) is dependent on "SCHEDULE POLICIES".

STEPS for a context switch: -

Step(1) => Decide whether to do a context switch(i.e. whether one of the four Situations arrived or not) and also decide, whether the context switch is permissible now or not.
Step(2) => Save the "CONTEXT' of old process.
Step(3) => Find the best process to schedule for execution using "PROCESS SCHEDULING ALGORITHM".
Step(4) => And restore "CONTEXT" of that best process.

Following code shows a possible code for "CONTEXT SWITCH", it is not real, hence termed as "PSEUDO - CODE".

```
if(save – context()) /* save the context of currently running process */
{
        /* pick another process to run */
        .
          .
          .
          .
          resume – context(new process);
          /*never gets back here */
}
/* The old process will execute from here */
```

The actual code for "CONTEXT SWITCH" in UNIX is supported to be the "MOST DIFFICULT" of all, because sometimes function calls do not return. Kernel may save context of a process at one point in the code and proceeds to execute some "OLD" process. When it resumes back to the proper process, it resumes execution according to the previously saved context of the process.

Here it becomes possible that return values may vary when kernel resumes the context of a new process and when it continues to execute context of old process. So it might become necessary to "HARD CODE" the program counter.

*** <u>Saving Context For Abortive Returns</u>: -** Some situations, mainly in sleep() and signal() algorithms, may arrive so that kernel must abort its current execution. In such situations kernel must suddenly change its context.

The actual algorithm of saving the context is setjmp() and the actual algorithm of resuming the context is longjmp(). The setjmp() algorithm stores the saved context in process's U area and continues to execute in the old context layer. When kernel wants to resume to the previously saved context, it uses longjmp(), which restores its context from U area.

*** Copying Of Data Between "USER ADDRESS SPACE" And "KERNEL ADDRESS SPACE": -** There are many occasions where data should be copied from "USER ADDRESS SPACE" to "KERNEL ADDRESS SPACE" and vice versa.

For this purpose many machines architecture allow kernel to access address in user's address space directly.

But kernel must ensure that the address, which is going to be read, or which is going to be written, must lie within the specified address space range. Otherwise page fault or exceptions occur.

*** <u>Manipulation Of Process's Address Space</u>: -** Many "SYSTEM CALLS" manipulate "VIRTUAL ADDRESS SPACE" of a process. In other words they manipulate regions.

The manipulation operations are....

- Allocating a region.
- Locking and unlocking a region.
- Attaching a region to a process.
- Changing size of the region.
- Loading a region.
- Freeing a region.
- Detaching a region from a process.
- Duplicating a region.

(1) <u>Allocating A Region</u>: - The algorithm is called as allocreg(). Allocation of a new region is done by the kernel during fork(), exec() and shmget().

[smhget() means shared memory get]

We already know that “REGION” is analogous to “INODE” and so that “REGION TABLE” is also analogous to “INODE TABLE”.

Thus as inode are arranged on “FREE LIST” and “HASH QUEUE”, the region is also has its own “FREE LIST OF REGIONS” and “ACTIVE LIST OF REGIONS”. [Here “HASH QUEUE OF INODES” is analogous to “ACTIVE LIST OF REGIONS”]. Both these data structures of a region are linked lists.

So free region is present on “FREE LIST OF REGIONS” and allocated region is present on “ALLOCATED LIST OF REGIONS”.

When kernel wants to allocate a region and wants to make entry in region table, it takes out first available region from “THE FREE LIST OF REGIONS” and after allocation puts it on “THE ACTIVE LIST OF REGIONS”. The allocated region is locked and then marked as either shared or private (local).

Now remeMBer that every process has its associated executable field called as program. This file is executed by the exec() call and program gets converted into process. So kernel sets this file’s inode as one of the entry in the region.(see contents of region). As inode is sharable, the process (whose inode this is) also shares its region via this entry of inode in the region.

Obviously as file’s inode is accessed, kernel increments this inode’s reference count. This is necessary to prevent other processes from accidental unlinking of this file.

Finally allocreg() algorithm returns a locked inode.

```
01 : algorithm : allocreg() /* allocate a region */
02 : input :(1) inode pointer(of the program file)
03 :        (2) region type(shared or private)
04 : {
05 :    remove first available region from “free list of regions”;
06 :    assign the region type as per the parameter;
07 :    assign “inode pointer” parameter to the “inode pointer” of region;
08 :    if(inode pointer is not null)
09 :            increment program file inode’s reference count;
10 :    place the allocated region on “active list of regions”;
11 :    return(locked region);
12 : }
13 : output : locked region.
```

Alive algorithm of allocreg() is quite analogous to ialloc() algorithm.

Look at the 8th line of algorithm, which is if(inode pointer is not null), this line is important because if inode pointer of program file is not null, then it indicates that some other process of same program’ instance is already running. Obviously the reference count of the program file’s inode is incremented.

(2) <u>Locking And Unlocking Of A Region</u> :- Similar to locking and unlocking of inode, region is also locked and unlocked. This is independent of allocating and freeing of region.

Thus kernel can lock an allocated region and then unlocks it with or without freeing it. Also it can lock a region in some important “PROCESS ADDRESS SPACE MANIPULATION” to prevent other processes to access the same region. After finishing its important task kernel unlocks the region as usual. E.g.: Page stealer process.

(3) <u>Attaching A Region To A Process</u>: - After allocation of region, now it should be attached to respective process. As like “allocation’, attaching a region tasks place in fork(), exec() and shmget() system calls. The algorithm is called as attachreg().

Attaching a region to a process means connecting the region to respective process's virtual address space. This region is either "NEWLY CREATED BY ALLOCREG ()" or "ALREADY EXISTING". Existing region means obviously "SHARABLE".

By allocreg() the region is now part of the "REGION TABLE". To attach it to the Process, we should make respective changes in that process's "PREGION TABLE".

So kernel creates a new pregion entry and then sets "TYPE" field of pregion entry to either Text or Data or Stack as per the region type.

Then it records the virtual address (recall that pregion entry has starting virtual addresses of Text, Data, and Stack regions of the process) of this region. This will allow kernel to determine "WHERE THE REGION IS EXISTED IN PROCESS ADDRESS SPACE".

Note: - The process must not exceed the maximum virtual address given by the system to it.

E.g. suppose a highest virtual address given to a process is at 8 MB location. So it will illegal to attach a region of 1 MB size (size of region is one of the content of region table) at location 7.5 MB. Because 7.5 MB + 1 MB = 8.5 MB which is beyond the maximum limit of "SYSTEM GIVEN" address space of 8 MB.

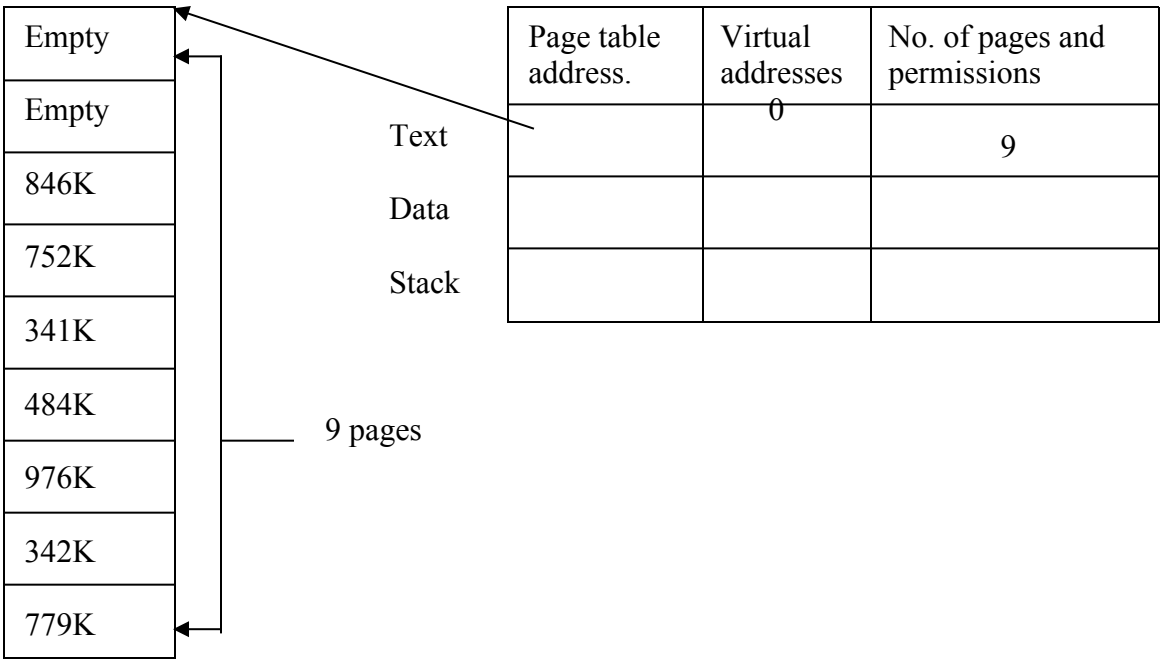
Also note that, the newly attached region's virtual address space must not overlap the addresses of existing regions of that process.

If attaching of region is legal (means neither overlapping nor extending beyond max), kernel then increments the "SIZE FIELD" of this process in the process table and also increments the "REGION REFERENCE COUNT" in region table.

```
01: Algorithm: attachreg() /*attach a region to respective process*/
02: Input: 1 pointer to the locked region, which we attach
03:        2 process (identifier) to which region is attached
04:                3 virtual address in process virtual address space, at which the region is going to be
attached.
05:          4 type of the region (text/data/stack/shared/private)
06  {
07:        Create a new "per process region table" entry for this process;
08:        In this new pregion entry:
09:          (a) set pointer to the attachable region,
10:          (b) set "TYPE" of region.
11:          (c) set virtual address of region (as recorded);
12:             check whether attaching of region is possible;
13:    increment "region reference count" in region table
14:    increment process's size (in process table) according to the newly attached region;
15:             Get & initialize new "register triple" for this newly attached region of the process;
16:    return (per process region table entry); /* pointer*/
17: }
18:   output: per process region table entry (pointer)
```

The attachreg() algorithm then gets and initialize the new "REGISTER TRIPLE" for this attached region. If the region is newly created (means not already existing), then "NEW PAGE TABLES" are required. This is done by growreg() algorithm. If the region is already exist (i.e. shared one), then it uses existing page table of existing region for newly got register triple.

Finally attachreg() algorithm returns pointer to "PER PROCESS REGION TABLE ENTRY".



Page Table

E.g. suppose there is an "EXISTING TEXT REGION", and we want address as 0, as text region points to "EMPTY" part, there are two "EMPTY" entries. As each entry is of 1 KB(we already assumed that one page is of 1 KB size),these are 2 KB space. So if new region size is below 2 KB 2 pages can be assigned here if the process desires.

But if the region size is of say 7 KB, then the pages will be assigned below 9^{th} entry in page table.

Note: - As region is contiguous space, the pages are not splitted. Means out of 7K, 2K are placed at 2 empty entries and remaining 5K are attached below 9^{th} entry, is never done.

4) <u>Changing Size Of A Region</u>: - a process may expand or shrink its virtual address space (but not beyond "SYSTEM GIVEN" max virtual address) by expanding or shrinking size of its regions.

Changing the size of a region is usually done by sbrk(). System call, which internally calls this growreg() algorithm.

When size of region expands, kernel makes sure the maximum virtual address space is not exceeded and growing region is not overlapping on existing other regions.

Note: - Shared region's is never changed by the kernel using growreg(). Rather it is impossible to grow or shrink the shared region, as it is accessible by multiple processes currently.

In other words growreg() can grow or shrink

- Non sharable (i.e. private) data region and
- Stack (which can grow automatically).

This can happen because both data & stack regions are private to the process while text region is usually shareable.

Thus text region and shared memory cannot grow or shrink, once they are initialized already.

When a region is to be grown, kernel first makes sure that physical memory is really available or not. Then it calls growreg() algorithm. If memory is not available then it tries to do some more measures, as we will explain in the chapter of "MEMORY MANAGEMENT". Then kernel allocates new pages in page table for new region or extends the existing pages. According to page table entries it assigns new register triple.

Note: - Only those pages of memory are assigned which do not support demand paging. Because demand paged pages are system resources and thus are kept reserved as possible as.

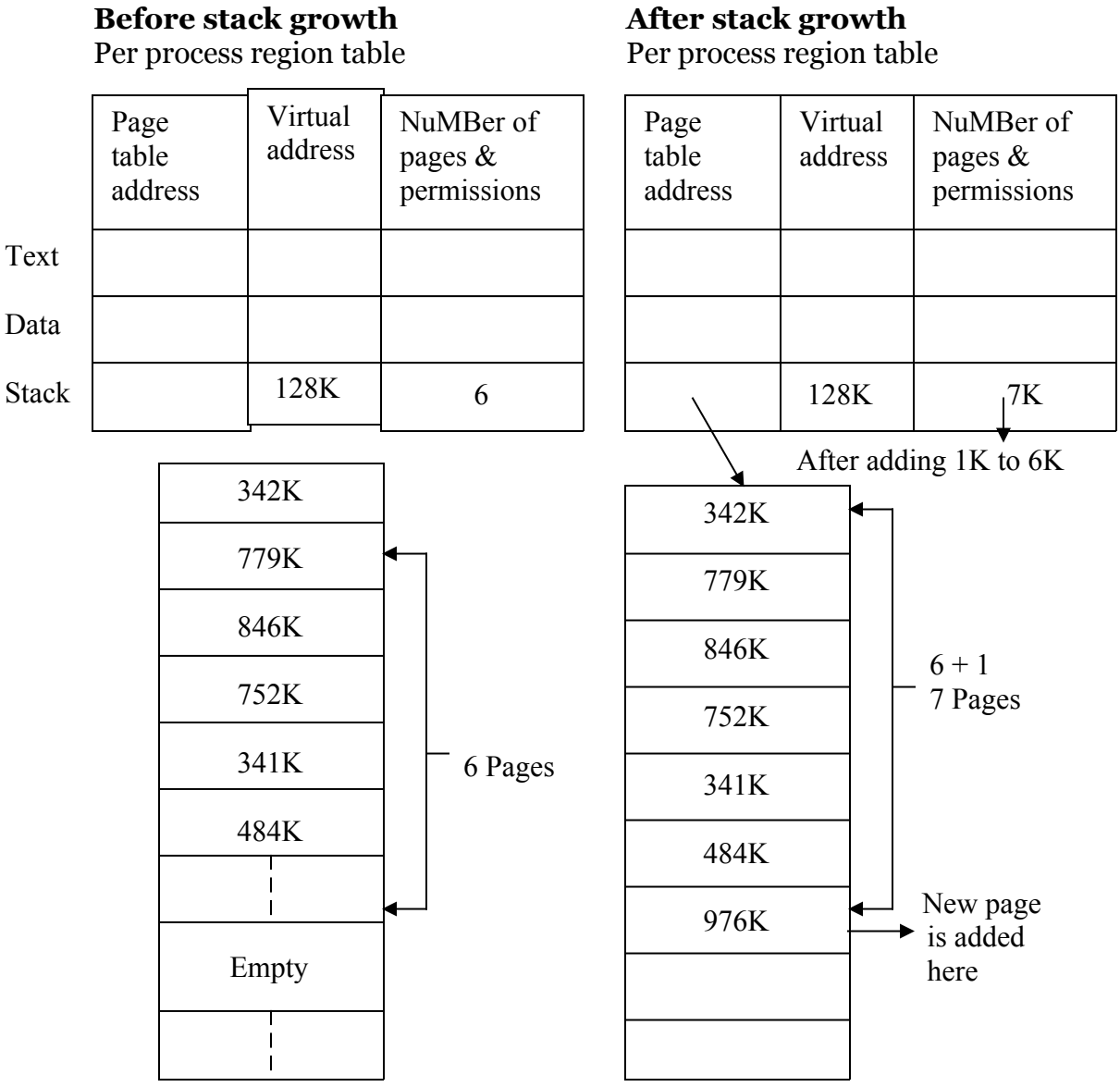
When a region is to be shrunken, kernel just releases memory & register triples assigned to the region.(not all but as per the required shrinkage)

In both above cases, kernel adjusts new size of virtual address space, new size of process table and new region size in region table. Initialization of pregion entry & register triples is as usual.

```
01: Algorithm: growreg() /* changes size of region*/
02: Input: 1 pointer to per process region table entry
03:            2 how much change in size (if +ve, then growing the region & if –ve, then shrinking the
region)
04: {
05:    if(region's size is increasing)
06:    {
07:            check whether increasing region's size is possible;
08:            allocate available(but not yet assigned) pages;
09:            if(system is not supporting "demand paging")
10:            {
11:                    allocate physical memory;
12:                    initialize memory tables accordingly;
13:            }
14:    }
15:    else /* means region's size is decreasing*/
16:    {
17:            free physical memory as appropriate;
18:            free pages & page table(memory) accordingly;
19:    }
20 :   according to above changes do consequent changes in page tables;
21:    set new size of region in region table & new size of process in process table;
22: }
23:  output: nothing
```

E.g.: - Suppose a stack region of a process starts at virtual address 128K and it is currently having 6 KBytes in it (means from 128K to 134K). Now suppose kernel wants to increase size of this stack region by 1 KB (i.e. by 1 page) and if currently there are 6 pages (means 6 K), then the stack is currently from 128K to 135K – 1 byte range. (135K – 1 byte means up to 138555).

So if the new size is acceptable and the range 128^{th} KB to 138555^{th} byte is not belonging to any other region attached to the process, then kernel extends the size of the region. Thus it makes new entries of pages in page table and initializes them by assigning new register triple



Growing the stack by 1K

(5) <u>Loading A Region</u>: - An operating system may or may not support demand paging.

If system supports "DEMAND PAGING", the kernel can "MAP" a file into the process's virtual address space during "EXEC" system call. This will be explained in the chapter of memory management.

But if system does not support "DEMAND PAGING", then the executable file is copied to the memory and the text data & stack (compiler given) are mapped as regions into the process's virtual address space.

While doing this, kernel may attach a new region at some different virtual address space (obviously somewhere in the process's virtual address space. Not outside of that). This creates "GAP" into the pages of page table.(as shown by 2 empty regions in figure on Page 149).

Sometimes programmers to create a gap between page tables programmatically explicitly use this feature. So when a user uses this program and tries to access such addresses (gapped) by pointer, then page faults are created deliberately. Such "EMPTY GAPPED" place in page table at address 0 creates page faults and abort.

The algorithm used for loading a region is called as loadreg(). This algorithm takes account of such "GAPS" between the virtual address space and attaches region and if required also expands the region accordingly.

Then it sets "STATUS" field in region as "BEING LOADED INTO THE MEMORY", and then reads data of this region from executable file and then copies into it. This reading is done by a special designed read system call (not same read() we observed up till).

If kernel is loading such a text region, which is "SHARED" by other processes too, then it is possible that some other process may use this region before it gets completely copied into the memory. In such situation setting of status "BEING LOADED" becomes helpful.

This kernel always checks "STATUS" of region, and if not yet "COMPLETELY LOADED", the process sleeps (The process, which want to use this region. Don't mistake it with the process, which is copying region's data from the executable file).

E.g. :- Suppose process A is loading a region and while loading it is new copying region's data from executable file into the region. While this reading & copying the process may sleep due to some reason. Now suppose process B tries to use that region (obviously because region is shared. Most commonly a text region), it looks for "STATUS" field in the region and if it is set to "BEING LOADED", process B also sleeps. When copying of process A completes, it sets "STATUS" field to "LOADED" and then process B wakes up and now onwards can use the region.

Obviously at the end of loadreg() system call, kernel changes this status flag from "BEING LOADED" to "LOADED" and then awakes all those processes which are waiting for this region to get loaded.

```
01: Algorithm: loadreg()  /* loads a portion of executable file into a region*/
02: Input: (1) pointer to per process region table entry
03:        (2) virtual address, where the region should be loaded
04:        (3) inode pointer for executable file
05:        (4) byte offset in executable file for starting point of region
06:        (5) byte count for amount of data to load.
07: {
08:     increase size of region (if needed) according to the size of region by using growreg();
09:    mark "status" field in region "being loaded";
10:    unlock the region;
11:    setup "u area" parameters for reading executable file;
12:              (a) target virtual address where the read data is put;
13:              (b) starting offset value from which reading is started;
14:              (c) number of bytes to read from executable file;
15:    read this desired portion of file into the region by using specially designed read system call;
16:    lock the region;
17:    mark "status" field in region as "completely loaded";
18:    awake all process, which are waiting for the region to get loaded;
19: }
20: output: nothing
```

We may think that "LOCKING THIS REGION" until completely loading will solve the problem and thus other process will not even try to access the same region.

But in above algorithm we rather unlock the region first. This will be clear in exec() call.

Now we will see one example of loading a region. Suppose a process wants to load its "TEXT" part of 7K into a region, which is attached to 0^{th} virtual address of a process. But process intentionally (i.e. deliberately) leaves an empty space of 1KB in page table (i.e. at beginning of region). We here assume that kernel already allocated entry of a region (by allocreg()) and attached this region to 0^{th} virtual address (by attachreg()).

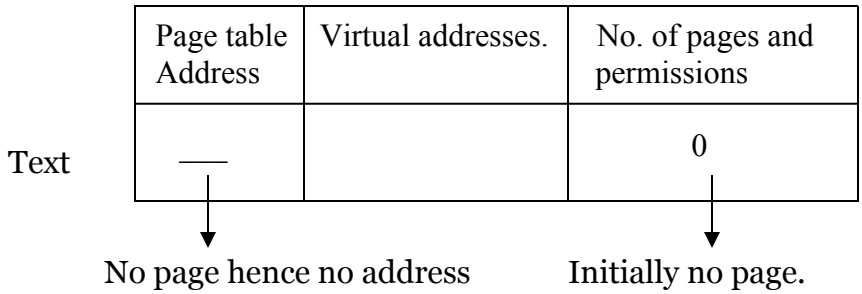
Now these are 2 tasks:

(1) Creating an "EMPTY" page space of 1 KB of 0^{th} virtual address.

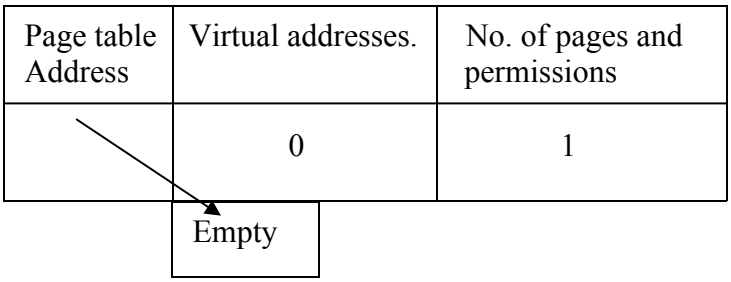
(2) Then attach 7 pages to load 7K-text region.

(a) <u>Original region entry</u>: -

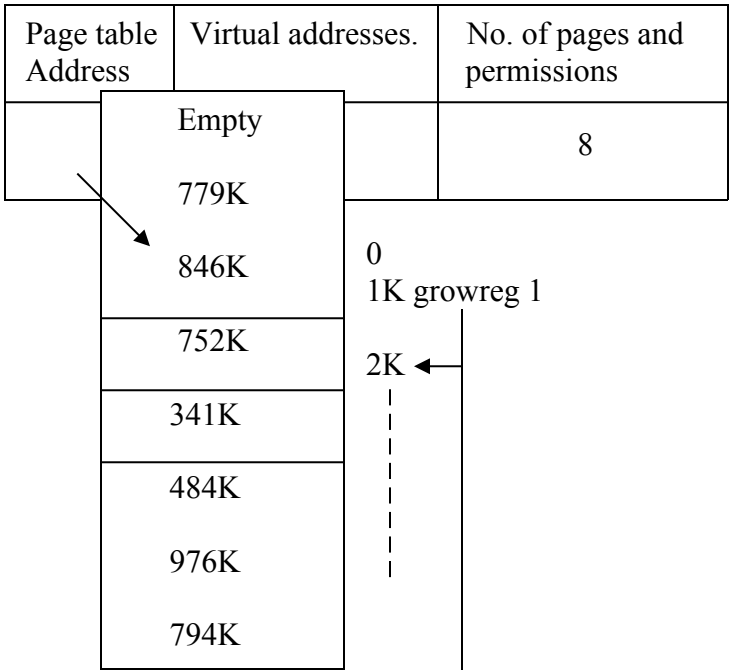
Per process region table.

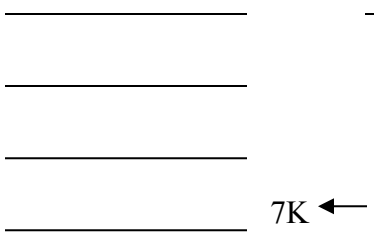


(b) <u>First growreg() call to Create "EMPTY" gap of 1KB</u> :-



(c) <u>Second growreg() call to create 7K pages</u> :-





To do these stacks loadreg() internally calls growreg() twice. First to create "EMPTY" gap of 1K at the beginning of region. Second to create page table entries of 7K, 1K of each page, started from 1K of page table (As 0th address is occupied by "EMPTY" entry of 1K).

(6) Freeing Of Region :- When a region is no longer attached to any process, the kernel frees the region and keeps it on "FREE LINKED LIST OF REGION", so that any other process can use it to reallocate. The algorithm for freeing the region is called as freereg().

```
01: Algorithm: freereg() /* frees an allocated region */
02: Input: pointer to a locked region
03:  {
04:    if (region's reference count is non-zero)
05:    {
06:              /* some other process is still using the region */
07:              release the lock of region;
08:              if(region is associated with inode of an executable file)
09:                        release the lock of inode;
10:              return;
11:    }
12:    /* if region reference count is 0 */
13:    if (region is associated with inode of an executable file)
14:              release the inode;
15:    free the physical memory still associated with region;
16:    free page tables associated with region;
17:    clear all region fields and make it empty;
18:    place this empty (freed) region on "free linked list of region";
19:      unlock the region;
20:  }
21:  output: nothing
```

First kernel looks for region's reference count. If it is non-zero then kernel knows that some other process is still using the same region. Hence kernel unlocks the region. Then it checks whether the region is associated with inode of an executable file(by looking in region's inode pointer field) and if it is, then it releases the lock of inode but doesn't release the actual inode. Finally the kernel returns.

But if the region reference count is 0, and if region is associated with inode of an executable file, then it actually releases the inode.

After dealing with the inode, kernel releases all physical resources (i.e. physical memory in the form of pages and page tables and register triples) and clears all fields in the region.

Then it puts this cleared, freed, empty region on "FREE LINKED LIST OF REGION" and unlocks it.

E.g.: - See the figure of "GROWING STACK REGION". Now suppose kernel wants to free this stack region, then kernel release 7 pages of physical memory and then page table & register triple.

(7) Detaching A Region From A Process: - Kernel detach a region from a process in exec(), exit() and shmdt() [*delete shared memory]* system calls. During detaching of region, kernel accordingly updates pregion entry and then cuts the "ATTACHMENT CONNECTION" between process & physical memory pages by invalidating the respective register triple. This also tells us that there is no effect on region itself. so region remains intact, and if shared other processes can see region as usual. Thus detachreg() and freereg() are different. Because in freereg() actually region is made empty and free.

The algorithm of "DEATCHING A REGION FROM A PROCESS" is called as "DETATCHREG()"...

```
01: Algorithm: detatchreg() /* detach region from a process*/
02: Input: pointer to per process region table entry
03:  {
04:    get page tables for this process;
05:    release them as appropriate; /*release table specifies to detachable region*/
06:   decrement process size in process table;
07:   decrement region's reference count field;
08:   if (region's reference count becomes 0 and region is not sticky bit)
09:              free region; /* algorithm freereg() */
10:    else /* means reference count is nonzero or sticky bit */
11:    {
12:          free inode lock if applicable; /* that inode which is concerned with region */
13:          free region lock;
14:    }
15 :  }
16:  output: nothing
```

In the algorithm, kernel gets respective page tables and releases those page tables, which are associated with the region. Then it decrements process size (as one region is removed) in process table and also decrement region's reference count.

If region's reference count touches 0, then it checks whether region (we will see about "STICKY BIT" later) is "TEXT REGION" and it is not "STICKY BIT". if both conditions are satisfied, kernel detaches & as well as frees the region by freereg() algorithm.

But if reference count is non-zero or if the region is "STICKY BIT", then it does not free the region. Instead it frees "INODE LOCK" of the inode, which is concerned with the region and also free the region lock. But **Note that,** region remains allocated (means not freed) and hence other process can use it as usual.

(8) Duplicating A Region: - The fork() system call duplicates region's of a process. But if a region is of "SHARED TEXT"(i.e. code of a program) or "SHARED MEMORY", then it won't really physically duplicate the region, but just increments the region's reference count this allows sharing of a region by a parent and its child.

When region is not "SHARED", then kernel must create physical copy of region (i.e. really duplicate) means kernel must...

- Create new region
- Allocate new region table entry
- Allocate new page tables and
- Also assign new register triple to the region.

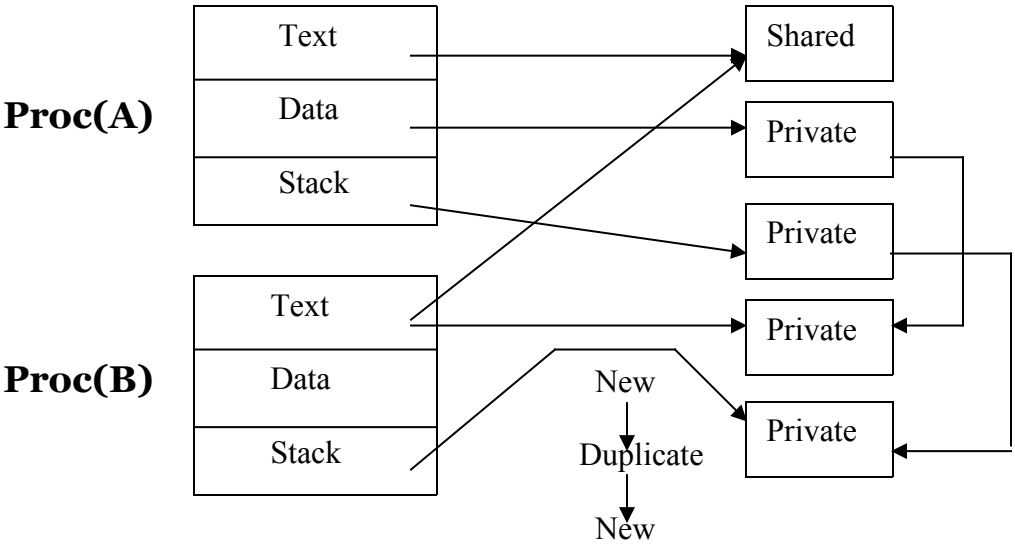
The algorithm is called as dupreg().....

```
01: Algorithm: dupreg() /* duplicates the existing region*/
```

```
02: Input: pointer to region table entry
03: {
04:    if (region "type" is "shared")
05:    {
06:          increment region reference count by using attachreg() algorithm;
07:          return (input region pointer);
08:    }
09:    else /* if region is "not shared " */
10:    {
11:          allocate new region using allocreg() ;
12:          setup page & page table & register triple according to input region;
13:          allocate actual physical memory;
14:          copy "input region contents" to "new region";
15:          return (pointer to newly allocated region);
16:    }
17: }
18: output: pointer to a new region, which is identical to the input region
```

E.g.: - Suppose process A forked its child process B and duplicates its regions. The "TEXT REGION" of A is shared so B need not to copy it physically. But data and stack regions are private to each other; hence they must be copied separately to new regions by duplicating them.

Note: - Though we said that "PRIVATE REGION" need to be copied as new region. It is not always necessary. We will see this in the chapter of "MEMORY MANAGEMENT".



Sleep: - This system call (or algorithm) changes the state of process from "KERNEL RUNNING" to "A SLEEP IN MEMORY". In contrast to this "WAKEUP" algorithm changes process state from " A SLEEP" to "READY TO RUN", either in memory or swapped. (See figure on Page 123)

Usually a process goes to sleep driving execution of a system call (and uncommonly driving execution of an interrupt).

Note that: - Process also goes to sleep when it faces a page fault (page fault is actually trying to access such a virtual address which is not actually physically loaded). So process goes to sleep while trying & retrying to read contents of the page (which are not there).

Sleep Events & Addresses :- In chapter 2 we already said that "PROCESS SLEEPS ON AN EVENT". Means process remains in sleep state until a specific event occurs. When event occurs, process wakes up and enters in the state of " READY TO RUN" either in memory or swapped.

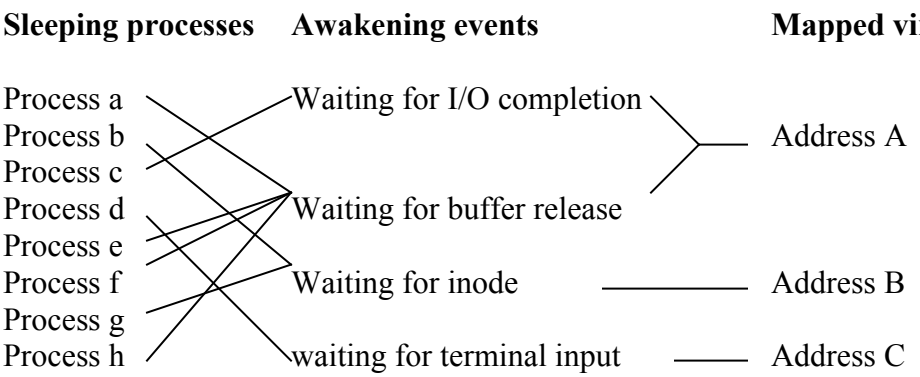
Though we said that a process sleeps on an event, internally event is nothing but a specific set of virtual address mapped by system.(i.e. part of kernel's virtual address space)

These address (i.e. events) reside in kernel. So system is only interested in that situation, when an event gets actually mapped with a particular address.

But in a multiprocessing system, we in advance, cannot know how many processes are waiting for the event. **So 2 things may happen…**

(1) When an event occurs and wakeup() call takes place, not only the process that we expect to wakeup, but all these processes sleeping on the same above event wake up, and enter into "READY TO RUN" state. Another thing also should be kept in mind, that suppose a "RESOURCE IS LOCKED" and hence many processes went to sleep waiting for that resource to get unlocked. Kernel never wakes up only one process at a time. Instead it just unlocks the resources & send message to all those processes slept on waiting for this event to occur. So at a time many processes may wakeup and enter into "READY TO RUN" state. Out of these, only one (The most eligible) process is scheduled to run and when this process runs, it again locks that resource (As process want to use this resource) and thus again remaining process go to sleep as their needed resource is locked again.

(2) Several events (rather more than one event) may map into one address. This statement is again a very important statement.



In above figure, these are 8 processes (a to h) are currently executing on a system. All are in "ASLEEP " stage. Process a, e, f, h are sleeping & waiting for a buffer to get released. Process b & g are waiting for inode to get released, process c is waiting for a certain I/O completion and finally process d is waiting for a terminal input.

As seen before events are mapped into addresses. So "WAITING FOR INODE" event is mapped to address B and "WAITING FOR TERMINAL INPUT" is mapped to the address C. this is ok! Because this is one to one mapping.

But sometimes it may happen (as shown in above figure) that 2 or more events get mapped into single address. E.g.: - In above figure event "WAITING FOR I/O COMPLETION" & event "WAITING FOR BUFFER" are mapped into single address, address A. so this is many-to-one mapping.

Now any of the above 2 events, when occur, all processes waiting for any of the above two events, will wakeup.

Means suppose I/O completes, so we expect only "PROCESS C" to get awakened. But not only process C, but all the processes which are waiting for a buffer to get released,(i.e. a, e, f & h) will also wake up.

Obviously process C when wakes up, gets the buffer and locks it again, a, e, f & h will again go to sleep.

But such mapping of many events to one address is rare and there is usually no chaos, because only related multiple events are mapped. Means in above example process is doing I/O on that buffer, which is the resource for processes a, e, f & h. so chaos is avoided by putting all others again to sleep when one is using the resource.

Algorithm of Sleep() :- The algorithm for sleep is as follows ----

```
01: Algorithm: sleep()
02: Input:(1) sleep address -> the address of event of which the process is slept
03:       (2) priority -> of sleep
04:  {
05:    raise "processor execution level" to block all intercepts;
06:    set "state of process" to sleep(in process table);
07:    put this process on "sleep hash queue" which is based on sleep address;
08:    save process's "sleep address" in process table;
09:    set "process's priority level" to "input priority";
10:    if (sleep of the process is NOT interruptible)
11:    {
12:            do context switch;
13:            /* when this process wakes up, it will start execution from here */
14:            reset processor execution level as before & allow interrupts;
15:            return (0);
16:    }
17:    /* if process's sleep is interruptible by signals*/
18:    if (no signal is pending against the process)
19:    {
20:            do context switch;
21:            /*when this process wakes up, it will start execution from here*/
22:            if(still no signal is pending against process)
23:            {
24:                    reset "processor execution level as before & allow interrupts";
25:                    return (0);
26:            }
27:     }/* the flow will come here only when the sleep is interruptible by signal and at least one is new
          received*/
28:    remove the process from "sleep hash queue", if it is still residing there;
29:    reset the processor execution level as before;
30:    if (process's sleep priority is set to catch signals)
31:    return (1);    /* 2nd parameter      i.e. below threshold */
32:    do longjmp() algorithm;
33 :  }
34: output :(1) it process's sleep is set to catch the signals (for its wake up), then the value returned is 1
                (2) or when long jump algorithm is executed as return status;
                (3) or 0 in all other cases.
```

- ✓ Kernel first raises the processor execution level to block all interrupts to avoid race conditions.
  Note that: - Kernel saves process's previous "PROCESSOR EXECUTION LEVEL"(before going to sleep), so that when it wake up, it could restore it back.
- ✓ Marks the "STATE OF PROCESS" as "A SLEEP" and makes appropriate entry in process table.

- ✓ Kernel maintains “A HASH QUEUE” of all currently slept processes. So kernel keeps this newly slept process on the hash queue. The hashing function is based on “SLEEP ADDRESS”. This hash queue is implemented using “LINKED LIST”.
- ✓ Kernel saves the “SLEEP ADDRESS”(i.e. parameter 1) in process table and also sets & saves process’s priority level to “INPUT PRIORITY” in process table. The “INPUT PRIORITY” allows the process to catch signals if process’s sleep is going to be “INTERRUPTIBLE“ by signals.
- ✓ In most cases, the process’s sleep state is not interruptible by any signals. In such cases, the process is going to sleep and hence “CONTEXT SWITCH” is allowed. So kernel saves context layer of this process (by setjmp) and does context switch to start context layer of another, scheduled process. Now process safely sleeps and when awakes, enter into “READY TO RUN” (either in memory or swapped). When scheduler schedules such process to run, the process returns from sleep state back to the sleep algorithm. (Thus the commented line is there).
  Note that: - Slept process has its entry in process table & hence may add some searching time, but slept process never-never consume CPU time.
- ✓ After coming back to sleep algorithm, kernel restores process’s previous “PROCESSOR EXECUTION LEVEL” to allow usual interrupts and returns 0 as its exit status of sleep algorithm.
- ✓ Above scenario is most common and a simple case of sleep algorithm. It is designed on a basic fact, that, a process is sleeping on such an event which is sooner/later going to occur always. i.e. event is sure to happen. Such as unlocking of buffer or inode, disk I/O, which are always locked temporarily. But a process, sometimes, may sleep on such an event, which is “NOT SURE” to get happened. And if it does not occur, then there must be a way to awake the process and continue its execution. For this purpose signals are designed. So when such “UN-SURE” event does not occur and process is in “A SLEEP“ state, the kernel immediately interrupts the process by sending a signal to it. Signal is designed in such a way that a process can recognize it even if process is sleeping. We will see about signals in next chapter. When we see the sleep algorithm, we may think that, this process’s sleep is “INTERRUPTIBLE BY SIGNAL” or “NOT”? This thing is the 2nd parameter. Which is the “PROCESS’S PRIORITY”.
  For sleep & wakeup, we must think about 2 priorities…
  (1) Sleep priority – which determines type of sleep.
  (2) Scheduling priority – based on sleep priority. I.e. whether “INTERRUPTIBLE BY SIGNALS” or “NOT”. Obviously above values are set according to the surety of event occurrence.

  There is a “SET PRIORITY” which is considered as “THRESHOLD VALUE”. So if the priority is set above of this value, then the process will be kept slept until the actual waiting even occurs. But if the priority is set below of this threshold value, sending signal to it can immediately awaken the process.
- ✓ Above discussion is necessary for second “IF BLOCK” in the sleep algorithm. This block tells us that kernel always checks “WHETHER THERE ARE PENDING SIGNALS” or “NOT” for this sleeping process.
  (a) If signal is pending, but “PRIORITY“ is set above the “THRESHOLD”, then process will not think about signal and will wakeup only when required event occurs. So it will be wake up by kernel with explicit wakeup call.
  (b) But if the “SLEEP PRIORITY” is below the “THRESHOLD”,then process actually does not go to sleep, but respond the arrived signal as if it is sleeping.
  Thus checking of “PENDING SIGNAL” before entering in sleep algorithm is must. Because if it is not done, the process may never wake up.
- ✓ Remaining things in second” IF BLOCK”(and its nested if block). Are similar to first “IF BLOCK” On completion of this second “IF BLOCK”, kernel returns 0.
- ✓ The 2nd “IF BLOCK”, checks the situation, when process’s sleep is interruptible by signals but “NO SIGNALS ARE PENDING” against the process.

Thus a third situation, where the process's sleep is interruptible by signal and if signal is arrived to wake up the process, then lines of algorithm which are not inside any "IFBLOCK" get executed.

Hence in such situation…

(a) Process is removed from "HASH QUEUE OF SLEEPING PROCESSES", if process is still there.
(b) Then "PROCESSOR EXECUTION LEVEL" is set to old processor execution level's value (which was the value before going to sleep)
(c) Perform long jump() algorithm to restore this process's context layer, if "SLEEP PRIORITY" is not set to catch signals.

Consider one example. Suppose a process went to sleep expecting some terminal input from first user. But user does not fulfill the requirement and left the workstation-keeping terminal in "IDLE" as it is. now second user arrives at his place and to restore the terminal, he shuts of the terminal.

Now the process, which was waiting for terminal input, must be awakened. As the "PROPER AWAKENING EVENT"(i.e. terminal input) does not occurred, this is "ABORTIVE ROUTINE", thus signal is sent to process, kernel restores the process's saved context layer by longjmp()(which was previously saved by setjmp()).

Note that: - Though not actually written in sleep algorithm, kernel saves current process's current context layer by setjmp(), while executing system calls & interrupts in anticipation that a situation like above example may occur.

As above routine is abortive, though it completes, it will return error.

Sometimes, same above situation occurs, means kernel wants to wakeup a process by sending a signal, but difference is that, here kernel does not want to do longjump() algorithm.

*Means process's "*SLEEP PRIORITY *" is set to catch signals. In such cases, bypassing of longjmp() occurs.*

In such situations kernel returns 1 and bypasses the longjmp() algorithm.

But to allow kernel to do this, a special priority value must be passed as 2nd parameter to sleep algorithm.

The return value (i.e. 1 here) can then be considered as " INDICATOR FROM KERNEL" to do necessary clean-up in abortive routines before calling longjmp() algorithm in wakeup.

Important point :- *"*CONTEXT SWITCH*" statements in sleep algorithm indicate that process is actually slept but when signal is present(i.e. statements not within any if block) then there is no "*COTEXT SWITCH*" so actually process is not sleeping, but it is receiving signal as it is sleeping.*

Wake up: -

"WAKE UP" wakes up a sleeping process. The algorithm is as follows ……

```
01:  algorithm: wakeup
02:  input: sleep address
03 : {
04:    raise "processor execution level" to block all interrupts;
05:    find "hash queue of sleeping processes" by using "sleep address";
06:    for(every slept process on this sleep address)
07:    {
08:            remove process from hash queue;
09:            mark process's state "ready to run";
10:            put process on "scheduler list" of those processes, which are "ready to run";
11:            clear "sleep address" field in process table;
12:            if(process is not yet loaded in memory)
13:                    wakeup swapper process;
14:            else if(awakened process is more eligible then currently running process)
15:                            set scheduler's flag accordingly;
```

```
16:    }
17:    restore "processor execution level" to original level;
18:  }
19: output: nothing
```

- ✓ The wakeup algorithm is handled by the kernel either driving the end of usual system call or during the end of an interrupt handler function.
- ✓ First kernel raises the processor execution level to block all interrupts to prevent race conditions.
- ✓ By using sleep address (which is parameter) it finds out the "HASH QUEUE OF SLEEPING PROCESSES".
- ✓ Then as explained, kernel never wakes up only one process. Rather it wakes up all those processes, which are slept on a "SLEEP ADDRESS"(an address which is mapped to single or multiple events as shown in figure).
  Thus kernel uses "FOR LOOP" here, with the testing condition "EVERY PROCESS WHICH IS SLEPT ON THIS SLEEP ADDRESS".
- ✓ Inside the loop, first it removes the process from hash queue.
- ✓ There it makes its "STATUS" as "READY TO RUN".
- ✓ Then it keeps this process on such a linked list, which is made up of all those processes, which are now in "READY TO RUN" state.
- ✓ Then it clears the "SLEEP ADDRESS" entry for this process in the process table.
- ✓ Now if the process is still not in memory then scheduler is not yet scheduled it. So kernel awakens the swapper process (assuming that process is swapped) and tells it to swap the process from "SECONDARY STORAGE DEVICE" back to "MEMORY".
- ✓ If the "AWAKENED PROCESS" is more eligible to run than the current process, then kernel sets "SCHEDULER FLAG" so that the process should go through "SCHEDULING ALGORITHMS", before returning to user mode.
- ✓ Finally kernel restores the processor execution level to allow all interrupts as before.

Note That: - Wakeup does not cause a process to be scheduled immediately to run, but it actually makes the process eligible to undergo scheduling algorithms.

Actually speaking, when a process arrives at a system call, kernel takes the charge and while executing code of system call suppose it finds that "A BUFFER IS LOCKED", so process sleeps on this event. Kernel now gives process, address of this "BUFFER RELATED EVENT" called as "SLEEP ADDRESS".

When kernel, eventually unlocks buffer, it is his duty to scan all process table entries and see for all "SLEEP ADDRESS ENTRIES" in process table. It takes out all those processes whose "SLEEP ADDRESS"(i.e. buffer related event) matches with "PREDEFINED EVENT ADDRESS"(which are part of kernel's virtual address space), and when match is found, it wakes up all those processes slept on waiting for buffer to get unlocked.