CHAPTER 7 PROCESS CONTROL

This chapter deals with those system calls, which control the process context. Thus system calls used by this chapter, internally use algorithms explained in last chapter.

- ✓ The fork() system call creates a new child process.
- ✓ The exit() system call terminates execution of a process.
- ✓ The wait() system call synchronizes parent's execution with child's exit.
- ✓ The exec() system call allows a process to invoke a "NEW" program by overlaying its address space with the executable file image of the program.
- ✓ The brk() system call allows a process to allocate more memory dynamically. (Including sbrk()).
- ✓ As wait() is used for synchronous operation, signals are used for asynchronous operations. exit() & wait() also depend upon signals. Almost all above system calls internally use sleep() & wakeup() algorithm. That is why we had seen them in last chapter. exec() system call, in addition, uses file system calls internally. Finally this chapter deals with "SHELL" & "BOOT" process of UNIX system.

System calls dealing with process's memory management.				**System calls dealing with process synchronization.**			**Misc**	
fork	exec	brk	exit	wait	signal	kill	setpgrp	setuid
dupreg	detatchreg	growreg	detachreg					
attachreg	allocreg							
	attachreg							
	growreg							
	loadreg							
	mapreg							

Creation of Process ! In UNIX system, the only way to create a new process is the fork() system call.

The process which invokes the fork() system call is called as parent process and the process which gets newly created as a result of fork() system call is called as child process.

The syntax for fork() system call is .....

```
pid = fork();
```

As seen in the syntax, this system call has no parameter. On return the two processes (i.e. parent & child) have identical copies of their user level context except for the pid.

Means in parent process pid is the child process ID (hence parent can identify its child by using this pid) but in child process, this pid is 0.

All... All processes created in UNIX system are forked, except the "PROCESS 0" which is created internally by the kernel at boot time. So "PROCESS 0" is the only process which is not created by fork().

While executing fork() system call, kernel follows following sequence of tasks.....

1) It allocates a new slot in the process table for its newly created child process.
2) It assigns a unique ID number (i.e. pid) to its newly created child process.
3) It creates logical copy of context of parent process and gives it to its child process. But while copying regions, it keeps "TEXT" region shared (By just incrementing "TEXT REGIONS" reference count) but creates separate physical copy of data & stack in its child process.
4) As parent is willing to give everything "OWNED BY IT" to its child, it also gives access to file table & inode tables of it to its child. Obviously as one new process is now accessing these data structures, their reference counts get incremented.
5) It returns pid of child process to its parent but returns 0 to the actual child process. So when parent says if(fork() == 0), it means that the code here onwards will be for the execution of

child process. The fork() system call has various "MEMORY RELATED" tasks. Hence code of its algorithm varies according to the system's memory management, i.e. whether swapping or demand paging.

Here we are going to follow the memory management scheme as of "SWAPPING" type. The algorithm of fork() is follows .....

```
01:  Algorithm: fork()
02:  Input: nothing
03:  {
04:    check for available kernel resources;
05:    get free process table slot and also get a unique PID number;
06:    check that user is not currently running too many processes;
07:    mark child process's state as "being created";
08:    copy data from parent's process table slot to newly allocated child's process table slot;
09:    increment counts of current directory inode & current root inode;
10:    increment counts of open files of parent process in "file table";
11:    make copy of parent's context (i.e. U area, text, data, stack) in memory;
12:    push "dummy system level context" into child system level context;
13:    this dummy context has such data that now child process can identify itself and when
14:    scheduler schedules it, it can run from here;
15:    if (executing process is parent process)
16:    {
17:            change child's state to "ready to run";
18:            return (child's ID as PID);
19:    }
20:    else /* means executing process is child*/
21:    {
22:            initialize its own "U area timing fields";
23:            return (0); /* to user*/
24:    }
25: }
26:  output: To parent process – child's PID number.
27:             To child process – 0
```

1) Kernel first makes sure that, it has enough available resources to call fork() successfully. Means in OS that supports "SWAPPING", kernel needs enough space either in memory or on disk to hold the newly created child process. In OS that supports "DEMAND PAGING", kernel needs page tables & associated register triples. If resources are not available, call to fork() fails.
2) Then kernel creates an empty slot in process table for its new child process, and also gets unique I.D. while assigning an unique ID number to the child process (which is later on identified as pid), kernel picks up "ONE GREATER THAN THE MOST RECENTLY USED ID NUMBER". If, sometimes, this new ID is already assigned to some process, kernel tries to assign the next higher ID number. If ID number reaches maximum value, assignment starts again from 0. But this is a very rare case.
3) Then kernel observes the process table and checks whether the user already has too many running processes or not.

   Limit on number of Processes: - The administrator configures the total number of maximum process at a time running, during system installation. This limit is for user. This limit on the user side is imposed to prevent user from acquiring all process table slots. Because if this happens for one user, other users can not create processes because process table is already full.

Also this is important that, no ordinary user is allowed to create such a process, which will acquire the last process table slot. Because if such thing happens no new process will be created and system will undergo deadlock! But super user has no such limits. He can create as much processes, as he wants. Also super user can occupy the last process table slot too. So for super user size of the "PROCESS TABLE" is the max limit of process table.

4) Now kernel starts coping data from parent's process table entry to the newly created child's process table entry. First it sets "STATUS" field in child's process table entry as "BEING CREATED". (State 8 in process table entry diagram).
5) **Note that: -** fork() creates environment for child process. It is our duty to fill this environment with a program execution by call to exec() so if exec() is not called, then fork() assumes that you want to run code next to fork() in parent program, as child. Hence many things from parent process get inherited to child. Thus child inherits parent's real & effective user ID numbers, parent process group and the scheduling priority values. Their scheduling parameters include initial priority value, initial CPU usage, scheduling times etc.
6) Due to the same above rule, child also inherits the parent's current directory & parents current root. As "CHILD PROCESS" is now a new process accessing current directory & root, inode reference count of "CURRENT DIRECTORY FILE" & "CURRENT ROOT FILE" are incremented each by 1. **Note that: -** If the parent process (or its any ancestor process) had executed chdir or chroot calls, changes are reflected to the new child process too.
7) Kernel also checks parent process's "USER FILE DESCRIPTOR TABLE" to know about how many files the parent process already opened. All these file descriptors (opened by parent) are inherited by child process. Thus "USER FILE DESCRIPTOR TABLE" of child process also gets all these file descriptor entries. As these file descriptors are shared, "COUNT" fields in their entries in "FILE TABLE" (i.e. global one) are incremented by 1. these changes are similar to dup(). But difference is that, dup() works for single process(in which it is called), while fork() works for two processes parent & child.
8) Now kernel creates child process's user level context. So kernel allocates memory for "U AREA" of child process. Also it allocates memory for regions, page tables for child process. Right now Text, Data & stack regions of parent & child are same. Hence kernel calls dupreg() to create identical but private physical copies of parent's data & stack regions in child process. But as text region is shared, it does not make physical copy of text region, instead it increments reference count of text region. After creating regions, kernel calls attachreg() to attach these regions to child process's virtual address space. Initially all contents of parent's U area are copied as they are to the child's U area. So contents of parent & child U area are identical but later on they may differ E.g.: - After completion of fork(), if parent opens a file, then its file descriptor will not be inherited by child because child process is already completely created and hence now child process is quite independent from the parent process.
9) The U area, regions & page tables are part of "STATIC CONTEXT" of the child process. Now its time to create dynamic context of child process. fork() is a system call, called in parent process. So when it is called, parent process enters in kernel mode and "USER LEVEL CONTEXT" of parent process is saved in stack. Now fork() is such a system call which lends to context switch from parent process to child process. Thus before leaving parent process, its context layer is saved. This "PARENT CONTEXT LAYER" contains register context & system context of parent process.
    a) Now recall from the topic of kernel stack, where we stated that location of kernel stack is implementation dependant. Means some UNIX flavors may keep kernel stack as a part of U area. If such a case, then while creating U area of child, all contents of parent U area are copied into child U area and hence kernel stack is also gets copied.
    b) But if some UNIX flavors keep kernel stack at some private location, then the story is different. Here kernel has to explicitly copy parent kernel stack to the desired private location of kernel stack of child. Whatever may be the case either (a) or (b), kernel stack of parent & child are absolutely identical initially. Later on they may differ as both

parent & child starts executing independently. So we don't have to worry about first context layer of child as it is created from parent process. But **Note that** after creating child process, flow goes back to parent again (returning pid to parent). Means now context switch is from child to parent.

Note that: - If child is really executing (which is not executing yet), then switching from child to parent will force kernel to create child's 2nd context layer. But right now child yet not started its independent execution. Hence kernel creates its dummy context layer, so that when control comes to child, it will again stat at fork().

The term "DUMMY" is used because kernel is creating context layer of such a process (i.e. child), which is not yet executing.

Within this dummy context layer, kernel sets register context of layer1, sets program counter etc. so that when required kernel should be able to restore the first context even if child is not yet really executed.

10) Now both "STATIC" & "DYNAMIC" context of the process are set. Thus now kernel sets "STATE" of the child process "READY TO RUN IN MEMORY" and child process's ID is returned to the parent (i.e. to the user). Here parent's fork ends. Now child process follows all usual scheduling algorithms and ready to execute, here child's fork ends, by returning 0 from the fork system call.

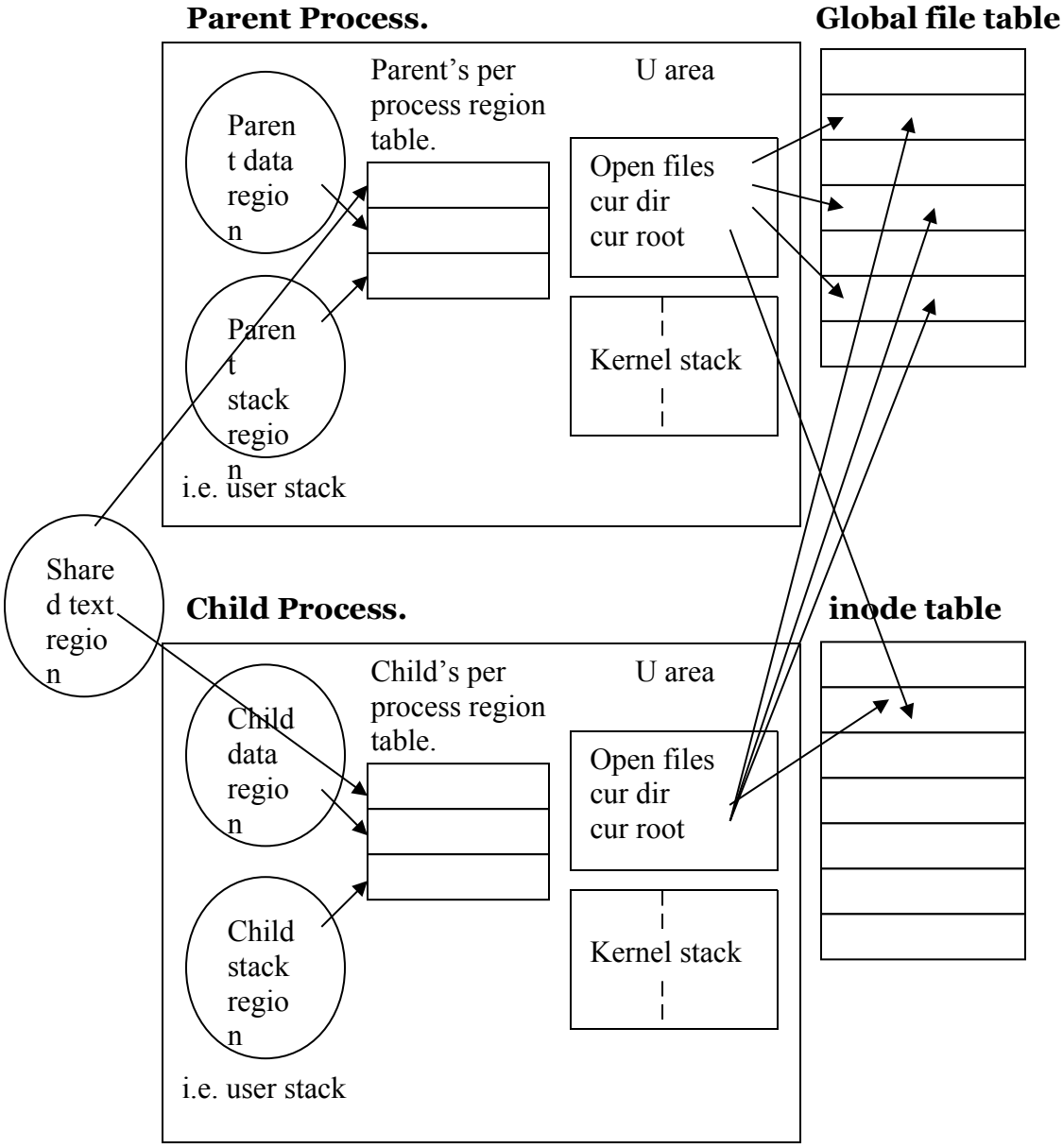


Figure on last page shows kernel data structures of parent and child processes after completion of fork() system call by parent.

- ✓ Both processes share files which parent had opened at the time of fork.
- ✓ So "FILE TABLE" entries for these shared files will have "COUNT" field greater than 1 of their previous values.
- ✓ Also notice that child process will have same current directory & current root as that of parent. Hence their (i.e. current directory & current root) inode reference count will be greater than 1 of their previous values.
- ✓ Also notice that, though parent data region & child data region and parent user stack region & child user stack region are identical respectively, as they are private, they have their separate physical existence.
- ✓ But "PARENT TEXT REGION" is shared between parent & child.

Example Program 1: - Now we will see one program to demonstrate how parent & child share file access.

```
#include  < fcntl.h>
/*global variables*/
int fd_read, fd_write;
char ch;
void main (int argc, char *argv[])
{
        /*code*/

        if (arc!=3)  exit(1);
        if((fd_read = open(argv[1], O_RDONLY)))
                exit(1);
        if((fd_write = creat(argv[2],0666)) = = -1)
                  exit(1);
        fork();

        /*both processes will execute same code onwards*/

        read_write() ;
          exit(0);
}

void read_write(void)
{
        /*code*/
        for(; ;)
        {
                if (read (fd_read, &ch, 1)!= 1)
                        return;
                write (fd_write, &ch, 1);
          }
}
```

After creating executable file of above program (i.e. after compiling & linking), this will become a program of "COMMAND LINE ARGUMENT TYPE".

So user is supposed to run this program from command line with 3 strings.

- First the program name. (i.e. argv[0]).
- Second will be first command line argument, which is name of the file to read. (Obviously existing) (i.e. argv[1]).
- Third will be the second command line argument, which is name of the file to write. (Obviously new always) (i.e. argv[2]).
- ✓ First number of arguments is checked, if no errors then proceed.
- ✓ Then existing file (i.e. argv[1]) is opened as read only. It no errors, then proceed.
- ✓ Then a new file is created with all accesses (i.e. –0.666). if no errors, then proceed.
- ✓ Now fork() system call is made. This will create a child process, and the code of 2 lines after fork()'s line, will be executed by both parent & child.
- ✓ A user defined function "READ_WRITE ()" is called here.
- ✓ After coming back, program will terminate with exit status "SUCCESSFUL" i.e. 0 [by exit(0)]
- ✓ In the body of user defined read-write() function, an "INFINTE FOR LOOP" (with two semicolons) is established.
- ✓ Inside the loop, read() system call reads from file descriptor fd_read, one character at a time in a global character variable ch. When it wont read a single character, return statement will end the loop and program flow will fall back in main(). While it doesn't fail to read 1 character, (means up till it can read one character), write system call will write to file descriptor fd_write one character at a time.

Now what happens due to fork()?

- ✓ The fork() system call creates a child process.
- ✓ Both parent & child will have their own virtual address spaces, hence both will run independently.
- ✓ Internally parent will copy its whole context into the context of child process. Hence initially both contexts will be identical.
- ✓ Both will access global variables as their own private copies. Thus values of fd-read, fd-write & ch will be private to both (***This is important because though above variables are global, here we are not talking about 2 functions in same 1 program. But in reality we are actually talking about 2 processes in 1 OS***). Their values will be different, because both are process.
- ✓ Both will have private copies of their stack variables (i.e. parameters) argc & argv.
- ✓ But also **Note that** the code upto the line of fork() is executed only by parent (as child is yet not exist) so parent opened fd-read & fd-write files.

But as we seen in the description of fork's algorithm, as kernel copies parent's U area "AS IT IS" to the child's U area and as U area has file descriptors, child inherits parent's file descriptors. Hence child will have access to the files opened by parent, by using same file descriptors.

Also **Note that,** *as file descriptors are same for parent & child, both will refer to same "*FILE TABLE*" entries and kernel already incremented the "*COUNT*" fields of these entries by 1.*

- As both processes run independently, the reading & writing of files will be never the same. Because after each read and write calls, file offsets in the U areas of both processes will differ.
- Thus contents in the "WRITE FILE" i.e. fd-write will depend upon which process is scheduled first by the kernel and also will depend on "CONTEXT SWITCH".
- If parent's read is followed by child's write or if parent's read-write pair is followed by child's read-write, then contents of source file (i.e. read file) and of target file (i.e. write file) will be identical.
- But suppose source file has a string "ab" and parent reads "a" from that string. Now if kernel does "CONTEXT SWITCH" to execute the child, before parent writes its read "a" to target file.

Now as per the context switch, child executes and suppose it reads "b" from the source file and writes it to the target file before parent gets re-scheduled.

So now target file has "b" written in it. Now again "CONTEXT SWITCH" occurs and parent gets re-scheduled. As parent is going to start its execution, where it was stopped, it writes its read "a" after "b" to the target file.

(***Here a question may arise, why parent cannot overwrite child's written "b" and place "a" in place of it. The answer is important. As "FILE TABLE" entries are shared between parent and child, when child writes "b" on target file, the byte offset field in file table entry of target file will be set to +1 from "b". Hence when parent writes "a", it is written after "b".)***

Thus though we expect that, if sever file contains string "ab", the target should also contain the same string "ab". But in above scenario, the target will have string "ba" written on it instead of "ab".

Example program 2: - Now consider another example of a program which uses pipe, dup, and fork together.

```
# include <string.h>
/* Global variable declaration.*/
char string[] = "Hello world";
/* Body of main() */
void main(void)
{
        /* Local variable declarations */
        int count, i;
        int ParentPipes[2], ChildPipes[2]; /* Pipes of parent and child */
        char buffer[256];
        /* Code */
        pipe(ParentPipes);          /* Create pipe for parent*/
        pipe(ChildPipes);           /* Create pipe for child*/
        if(fork() == 0)
          {
            /*This is child process execution.*/

           close(0);                          /*Close stdin*/
           dup(ChildPipes[0]);               /* stdin will become child's read pipe */
           close(1);                          /*Close stdout*/
           dup(ParentPipes[1]);              /* stdout will become Parent's write pipe */

           /*Close un-necessary pipe descriptor */

           close(ParentPipes[1]);            /* No harm, because already duplicate as stdout.*/
           close(ChildPipes[0]);             /* No harm, because already duplicate as stdin */
           close(ParentPipes[0]);            /* Not used*/
           close(ChildPipes[1]);             /* Not used*/

           for(;;)                            /* In finite loop*/
              {
               if((count = read(0, buffer, sizeof(buffer))) == 0) exit(1);
              write(1, buffer, count);
           }
```

```
        }

        /* This is parent process execution */
        close(1);                       /* close stdout */
        dup(ChildPipes[1]);             /* stdout will become child's write pipe*/
        close(0);                       /* close stdin*/
        dup(ParentPipes[0]);            /* stdin will become parent's read pipe */

        /* Close un-necessary pipe descriptor */

        close(ChildPipes[1]);           /* No harm because already duplicated as stdout */
      close(ParentPipes[0]);          /* No harm, because already duplicated as stdin */
        close(ChildPipes[0]);           /* Not used*/
        close(ParentPipes[1]);          /* Not used*/

        for(i=0;i<15;i++)               /*finite loop of 15 iterations */
      {
          write(1, string, strlen(string));
          read(0, buffer, sizeof(buffer));
        }
}
```

- As a rule, child inherits standard input(0) and standard output(1) of parent file descriptors.
- Call to pipe() system call creates 2 more file descriptor one for parent and one for child in the array. Note that 0th element is "READ PIPE" and 1st element is "WRITE PIPE".
- fork() system call creates a child process having initial context same as that of parent. Both processes will run independently.
- Variable access to both these processes is same as explained in previous example.

- **Now see the code of parent process.** First it closes its standard output descriptor (i.e. 1) by calling close() system call.

  Then it call dup() for child's "WRITE PIPE" i.e. ChildPipes[1]. As dup() causes kernel to select first empty slot in file descriptor table, it will choose 1st slot(indexing from 0) which was closed on previous line of code. So in other words parent's stdout will be child's "WRITE PIPE". So whatever written on parent's stdout will be written on child's write pipe. Thus here parent will send data to the child during its execution when parent is scheduled, and when child gets scheduled it will read that data though its read-pipe (i.e. ChildPipes[0]);

  Then parent calls close() again to close standard input descriptor and replaces it by parent's read pipe(i.e. ParentPipes[0]) by dup() system call. So in other words parent's stdin will be its own read pipe. So whatever it will read from standard input it will actually read contents of its read pipe(i.e. ParentPipes[0]) whenever it will get scheduled.

- **Now see the code of child process.** The statement if(fork() == 0) ensures that the code henceforth will be for child. Because recall that fork() returns 0 to child and pid to parent.
  First child closes its standard input by close(0) call and replaces it with its own read pipe. In other words, child's stdin will be its own read pipe. Thus whenever child process gets scheduled, whatever it will read from its standard input, it will actually read from its read pipe (i.e. ChildPipes[0]);
  Then it closes its standard output by calling close(1) system call and replaces it with parent's write pipe (i.e. ParentPipes[1]). In other words, child's stdout will be parent's write pipe. Thus

whenever child gets scheduled, whatever it will write, it will go to the parent's write pipe and when parent gets scheduled, parent will be able to read those contents via its read pipe. This is the way by which child sends data to its parent.

- By above two codes parent-child will send data i.e. messages to each other establishing Inter Process Communication (IPC);
- Also notice how parent and child close their un-necessary pipe descriptors.
    - Parent closes its read and write pipes as they are already converted to parent's stdin and child's stdout respectively.
    - As parent's stdout and child's stdin are not necessary here (Though necessary in child's code) they are also closed.
    - Child takes similar actions on its un-necessary pipe descriptors.
    - Notice the order in which pipe descriptors are closed. i.e. parent first closed child's pipe descriptors and child first closed parent's pipe descriptors, in the order of their replacements. Closing of these un-necessary pipe descriptors is a very good programming practice, because
        - ⇨ As only stdin, stdout and stderr are used, there is no danger of "SYSTEM's FILE DESCRIPTOR LIMIT" to get overflown.
        - ⇨ As no other file descriptors and pipe descriptors are there, it gives clean environment for both processes.
        - ⇨ Closing of write pipe allows other process to get end-of-file in its read pipe. If this is not done, menus if "WRITE PIPE" is not closed, "READ PIPE" will never able to know when reading is to be finished. Similar is there for closing of read pipe. But it will be ok if closing of read pipe is forgotten. Because, if there is no data to write from read pipe, there is chance that named pipe will return error. (See in pipe)

Note that *above program will not work correctly if child forgets to close its "*WRITE PIPE*" ChildPipes[1], before entering its "*FOR LOOP*".*

Both are reading from "0" means from stdin and both are writing on "1" means on stdout.

The output of above program is quite unpredictable, because it will depend upon which process is scheduled in respect to others.

- If child executes its read system call before parent's write, the child will sleep because pipe is empty and there is no data to read for the child. After child's sleep, the parent will be scheduled, which will write data into child's pipe and will then wake up the child, which now onwards can read the data from its pipe easily. Because read of 0 means stdin, which is mapped to child's read pipe, whose write pipe is mapped to parent's stdout.
- If parent writes on its pipe, before child reads means parent writes on ParentPipes[1], which is mapped to child's stdout. Thus next call to "READ" in parent's loop will not complete until child reads its stdin and writes on its stdout.
- Above scenario are only at the beginning. After that order of execution is fixed.
  Means each process will complete its read and write and will not complete its next "READ" until other process completes its read and write.
- The parent will exit after 15 iterations, where child will encounter EOF of its read pipe because there is now no writer process (as parent ends after 15 iterations)
- But if child wants to write on pipe, even though parent does not exist, it will receive a signal that there is no reader and hence there is no meaning of writing on pipe hereon words. This will be notified to the child via signal.

Signals: - Signals are the entities which inform the process / processes about occurrence of asynchronous events.

Either processes can send signal to each other or kernel can send signals to process / processes.

Processes use kill() system call to send signals to each other.
There are such 19 signals in system V release 2. These signals are classified into 7 types: -

(1) Signals concerned with termination of a process.

⇨ **Such signal is sent, when….**

- Either a process exits.
- Or when a process calls signal() system call with "DEATH OF CHILD" as its parameter.

(2) Signals concerned with process induced exceptions.

⇨ **Such signal is sent, when….**

- Either a process tries to access an address, which is out of its virtual address space.
- Or when a process tries to write on such a memory location, which is actually "READ ONLY" (such region is usually program's text region).
- Or when process executes such a privileged instruction, which is actually for some hardware errors.

(3) Signals concerned with unrecoverable conditions during execution of system call.

⇨ **Such signal is sent, when….**

- System resources are finished.
- Or when a process calls exec() even though its address space is already released.

(4) Signals concerned with unexpected error conditions during execution of a system call.

⇨ **Such signal is sent, when…**

- A process is calling such a system call which is not existing.
- A process is writing on a pipe and such pipe is not having reader process to read it.
- A process is giving such a byte offset to lseek() system call, which is not existing in the file.

Here we may think that instead of signal, displaying error will be more logical. But Dennis Ritchie said, "*USE OF SIGNAL IS MORE CONVINIENT FOR SUCH SYSTEM CALLS WHICH DO NOT CHECK FOR FAILURE OF SYSTEM CALL.*"

(5) Signals concerned with processes in user mode.

⇨ **Such signal in sent,when…**

- A process in user mode wants to receive an alarm signal after a specific period of time. (i.e. setting timer).
- A process in user mode wants to send a signal to other process by using kill system call.

(6) Signals concerned with terminal interaction.

⇨ **Such signal is sent, when…**

- User hangs up a terminal.
- Or user presses "BREAK" or "DELETE" keys on terminal's keyboard.

(7) Signals concerned with "TRACING" of execution of a program.
Such signal is sent during "DEBUGGING" of a program.
There are many important points regarding signals, such as…

- How kernel sends a signal?
- How kernel handles a signal?

- How process reacts to a signal?
- How process handles a signal?

➢ **How kernel Sends a signal**: - Recall from "CONTROLS" of "PROCESS TABLE" that, process table has a "SIGNAL FIELD" in it which is for those signals which are sent to process already but yet not handled by the process.

So when kernel wants to send a signal to a process, kernel first finds out "PROCESS TABLE SLOT" of that process and sets a bit in that field. This setting is according to the type of signal.

Now recall from the topic of "SLEEP" that, if the process's sleep is set as "INTERRUPTIBLE" by signals, then kernel sends signal to the slept process, interrupt its sleep and awakes it.

Here job of the signal sender (whether kernel or even a process) finishes.

➢ **How kernel handles a signal?**

- It is very important to note that, kernel handles a signal for a process, only when process returning from kernel mode to user mode.

➢ **How process handles a signal?**

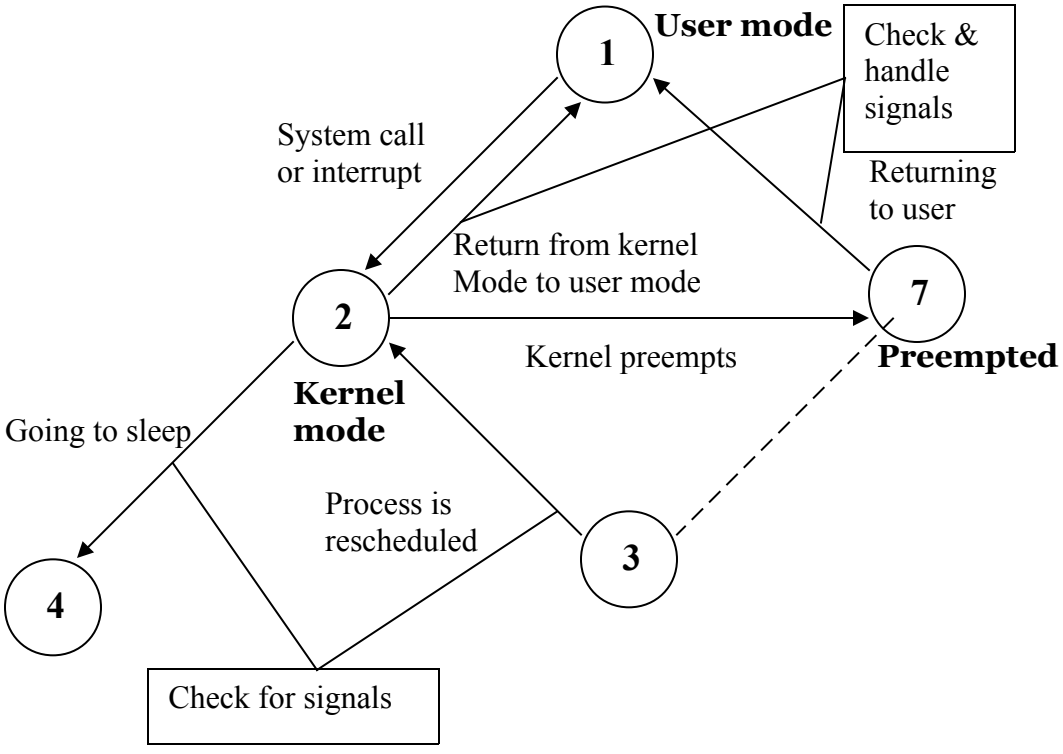
- A process can remember different types of signals. But it has no memory of how many signals of a particular type it received. E. g. :- If a process receives a "HANGUP SIGNAL" and a "KILL SIGNAL", then it can set the bit in process table's signal field about "HANGUP SIGNAL" and "KILL SIGNAL". But it cannot tell how many "HANGUP SIGNALS" it already received. Neither can it tell how many "KILL SIGNALS" it received.

  Below diagram is "CUT PORTION" of "PROCESS STATE TRANSITION DIAGRAM" which shows checking and handling position of signal.

  As mentioned before kernel can handle signals only when process return to user mode (either from kernel mode or from preempted situation).

  But kernel can check for signals at 2 locations.

  - When process enters in sleep or returning from sleep (i.e. gets rescheduled)
  - And when process returns from kernel to user mode.



Above description tells us that a process receives signal at different location and it is handled at another location. Obviously we can arrive at a conclusion that signal does not have a quick effect on the process which is running in kernel mode. Because neither receipt nor handling can be done in kernel mode.

So if a process is running in user mode and kernel is right now handling an interrupt, and if that interrupt sends a signal to process, kernel will not handle it readily. Rather it will complete its interrupt handling and then it will recognize and handle the signal when it returns from the interrupt.

So obviously process can never execute in user mode before handling "THE ALREADY PRESENT" (i.e. outstanding) signal.

- **How process reacts to signal?** To react to a signal process has one of the three choices...
  - ! Either process will exit as a reaction to a signal.
  - ! Or process may ignore the signal.
  - ! Or process may execute a particular user defined function as a response to receipt of signal.

To determine whether a process really received a signal or not, kernel executes an algorithm called as issig() (which means "**is signal**?").

```
01: algorithm: issig() /* Test for receipt of signal */
02: input: nothing
03: {
04:    while("received signal field" in "PROCESS TABLE" is not 0)
05:          {
06:                find a signal number sent to the process;
07:                if(signal is "death of a child")
08:                      {
09:                            if(ignoring "death of child" signal)
10:                                  free process table entries for zombie children;
11:                            else
12:                            if(catching "death of child" signal)
13:                                  return(true);
14:                      }
15:                else
16:                if(we are not going to ignore other signals)
17:                      return(true);
18:                turn "off" signal bit in "received signal field" in "PROCESS TABLE";
19:          }
20:   return(false);
21: }
22: output : true – if process received a signal that is not ignored.
             false – otherwise (i.e. either signal are ignored or there are no signals)
```

- Here kernel looks inside the process table for "RECEIVED SIGNAL (but yet not handled) field". If this field is nonzero, then it is an indication that signal is received.
- From this field it takes the signal number.
- If signal number corresponds to the signal "DEATH OF CHILD", then it takes 2 actions according to the reaction to the "DEATH OF CHILD" signal.
  Means, if this signal is to be ignored, then kernel just frees the process table entries for "ZOMBIE STATE" children.
  But if this signal is going to be catch, then kernel returns true.

- For other all signals (besides “DEATH OF CHILD”), if they are not going to be ignored, kernel returns true.
- But if all signals (except “DEATH OF CHILD”) are going to be ignored, then there is no meaning to keep the “RECEIVED SIGNAL FIELD” bit set. So it turns that bit “OFF” and returns false.
- So ultimately if received signal is not ignorable, kernel returns true or otherwise false. Means it will return false even if the “RECEIVED SIGNAL FIELD” in process table is 0.

Note that: -

- ⇨ “IGNORANCE” or “NON-IGNORANCE“ will depend upon the process’s “WILL” in user mode and upon the “SLEEP PRIORITY” setting for slept process. (See “SLEEP” for more details).
- ⇨ Also note that, for “NON-IGNORABLE” signals, kernel is not doing something drastic. It just notes the existence of non-ignorable signal by returning true.

Handling Of Signals: - Though it seems that a process, which receives a signal, is the process, which handles it, the actual story is different. Actually kernel handles the signal on behalf that process. Hence it is said, “IT IS THE KERNEL WHO HANDLES THE SIGNAL IN THE CONTEXT OF THE PROCESS THAT RECEIVES IT.”

Obviously the process which receives the signal must be in running state allow the kernel to handle that signal on behalf of it.

As stated before these there are 3 cases for handling signals….

(a) The process exits on receipt of signal, or
(b) The process ignores the signal, or
(c) The process executes a particular user defined function on receipts of the signal.

Out of above 3 cases the default is the first one. Means process gets exited on receipts of signal in kernel mode.

But if other two cases to happen, then process must specify special action (out of the remaining two) with the signal() system call.

The syntax of signal() system call is…

oldfunction = signal(signal-number, new-function);

Where signal number is the signal’s number for which the process wants
to specify its special action. The new-function is the particular user defined function that process wants to executes on receipts of the signal, and the old function is the value (i.e. address) of the function that was executed by the process in past when the same signal was dealt.

The new function parameter is the name of function in string form (recall that name of a function is itself address of that function in c).

But if you wish you can specify 0 or 1 instead of function name in 2nd parameter to signal() system call.

If you specify 0 (which is the default if unless specified), then kernel assumes that you want to exit the process in kernel mode on receipt of the signal. (i.e. (a))

If you specify 1, then process will (and also the kernel) ignore all future and present occurrences of the signal. (i.e. (b)).

And if you specify “ADDRESS OF NEW FUNCTION” (i.e. name of the function), then process will execute that function on receipt of this signal. Such function is called as “SIGNAL CATCHER FUNCTION” or simply the “CATHER FUNCTION”.

The “U area” of the process contains array of signal number and associated catcher functions. (See “contents of U area”, the 11th content).

So whenever new signal arrives and if process gives user defined function for that signal, kernel stores number of this new signal and its associated specified user defined function in that array. **Note that** handling one type of signal has no effect on handling of other types of signals.

The algorithm for "HANDLING SIGNALS" is called as psig() which is as follows.....

```
01 : algorithm : psig() /* Signal handling algorithm after signal is detected by issig*/
02 : input : nothing
03 : {
04 :   get the signal number which is set in the "process table entry";
05 :   Now clear the signal number in process table entry; /* As we are going to deal with it now*/
06 :   if(user had called signal() system call to ignore this signal)
07 :                 return;        /* done */
08 :    if(user had called signal() system call with "user defined function" or if signal is already
        associated
          with "user defined function" in the "U" area array)
09 :          {
10 :                 get virtual address of that "signal catcher function" from the "U area" array;
11 :                 clear "U area" array entry of the address of that signal catcher function;
                      /* This line creates undesirable side effects */
12 :                 modify the user level context :
13 :                 means artificially create user stack frame for the call to signal
14 :                 catcher function;
15 :                 modify the system level context :
16 :                 means write address of signal catcher function into the "program counter" field of
                      saved register context;
17 :                 return;
18 :          }
19 :   if(signal is of such a type that system should make "core dump image" of the process)

20 :          {
21 :                 create a file named "core" in the current directory;
22 :                 write contents of "user level context" into above file;
23 :          }
24 :          execute "exit" algorithm immediately;
25 : }
26 output : nothing
```

- Kernel first does the tasks of issig algorithm. Means it determines signal type, turn "OFF" the appropriate signal's bit in the process table entry (which was set when signal was received). In other words it does the tasks of first two statements in the algorithm. Means it gets respective signal's number from process table entry and then clear signal number in that entry.
- If process receives such a signal which was selected to be ignored (i.e. signal() system call with 1 at the function parameter), then kernel returns and proceeds as if signal is never received. As kernel does not reset the field in U area of the process (i.e. the field which shows the signal is ignored- 11^{th} content) about the signal, if this signal re-arrives, process re-ignores it due to these lines of algorithm.
- If process had decide to catch the received signal, (means process already called signal() system call with particular user defined function name as 2^{nd} parameter), then it executes that user defined function. But note that this execution of user-defined function is done when....

*** Kernel does following 5 steps....**

Why to make so many things? After coming in user mode, we except that "SIGNAL CATCHER FUNCTION" is executed immediately. But we had not called signal catcher function explicitly. So kernel takes name of this function from signal() call and executes it by these steps.

a) Kernel first accesses the saved register context of this process (which was previously saved for return to user mode) and finds out program counter and stack pointer in it.This "SAVED REGISTER CONTEXT" has return address of system call in program counter.
b) It clears the signal catcher function's field in the U area's "SIGNAL ARRAY". This statement has some undesirable result. We will see about it later.
c) Then it sets above "CLEARED FIELD" to its default state. (i.e. to exit state).
d) Now kernel creates a "NEW STACK FRAME" and writes program counter and stack pointer (retrieved in step (a)) in it. Here it writes the return address of system call. (i.e. address of that system call witch moves process from user mode to kernel mode).

   Now the user's stack looks as if the process had called the "SIGNAL CATCHER FUNCTION" (though process yet not called it) at that point where process had made the system call where the kernel had interrupted the process before recognition of the signal.
e) Now kernel modifies the saved register context (i.e. accessed in step(a)). As seen in step (a), program counter and stack pointer had some different values. But here kernel changes program counter to the address of the signal catcher function and also changes the stack pointer to consider the growth of stack which occurred due to the addition of new frame in step(d).

   ⇨ After doing above 5 steps, now kernel allows process to return to user mode, where it is ready to handle the signal. The "RETURN" statement states this step.

Thus now process comes to user mode at last and starts its execution from the "NEW ADDED STACK FRAME" and from "THE CHANGED REGISTER CONTEXT". Obviously as program counter is pointing to the address of "SIGNAL CATCHER FUNCTION" and "STACK POINTER" is pointing to the "NEW ADDED STACK FRAME", it has to immediately execute the "SIGNAL CATCHER" function because by steps (d) and (e) it returns to the place where it entered in kernel mode (i.e. before returning to user mode it was in kernel mode) due to some system call or due to some interrupt originally occurred.

- That is all about that signal which process wants to catch by "SIGNAL CATCHER FUNCTION".
- Now consider those signals for which "0" is specified in 2nd parameter (i.e. at the place of signal catcher function) of signal() system call.

  As explained before for such signal's kernel exits the process.
  Now such signals are of 2 types…
  (1) Signals due to some "PROBLEMS" in the process.
  (2) Signals, where there are no "PROBLEMS" in the process but user wants to exit the process "PREMATURELY" by pressing "DELETE" or "BREAK" key on the terminal's keyboard.

How To Handle Signals Of Type (a): - When process has some problems and hence kernel generates some signal, then kernel gives programmers a chance to see what happens with the process. This is quite helpful for debugging the program. (in programmer's terminology this is called as assertions).

So before exiting from such process, kernel makes dump of "CORE" image of process and writes "CONTENTS OF USER LEVEL CONTEXT" to a file called as "CORE". After that programmers may see this file to find out the cause of exit.

Example of such signal: - i.e. when a process executes an illegal instruction or when a process accesses an address, which is outside of its virtual address space.

And then executes exit algorithm to actually exit the process.

How To Handle Signals Of Type (b): - Sometimes there is no problem with the process. But user many want to terminate the process prematurely by pressing "DELETE" or "BREAK" on terminal's keyboard.

As in this case, there is nothing wrong with the process, kernel does not create "CORE DUMP" and just exits immediately.

Sometimes, there is nothing wrong with the process, but still user wants its "CORE DUMP", then quit signal is used (either in the program or from outside the program) by typing ctrl + | character combination. This feature is helpful when a process get caught in infinite loop.

Example: - Now we will see one example of such a program which wants to catch a signal (say interrupt signal) and also wants to handle the signal by using "SIGNAL CATCHER FUNCTION".

```
# include <signal.h>
/* Body of main */
void main(void)
{
        /* Function declaration */
          void catcher(void);
        /* code*/
        signal(SIGINT, catcher);
        kill(0, SIGINT);
}
void catcher(void)
{
          /* Body of catcher function */
}
```

The "INTERRUPT SIGNAL" is denoted by a constant "SIGINT".

- By signal() call program wants to catch the signal of "SIGINT" and wants to handle it by "catcher()" function.
- By using kill() system call it is sending on "INTERRUPT" signal to itself . The "0" in place of first parameter ensures that the "SIGINT" signal will go to all those processes, which belongs to this process's group. Obviously it will arrive at this process too.
- When kernel executes signal(), it comes to know the type of signal that process wants to handle and also knows about the "SIGNAL CATCHER" function.
- Now kernel executes kill() system call. As it is a system call, it will first go to library and library creates "OPERATING SYSTEM TRAP". This gives kernel the "ENTRY POINT" of kill() system call in its IVT say at "AddOfEntryPointOfKill" address. (The address is actually hexadecimal, but for simplicity we took it as name). And suppose the return address from kill() is "AddOfReturnOfKill" address. Also suppose that the address of the "SIGNAL CATCHER FUNCTION" (i.e. catcher() in above code) is "AddOfCatcher".
- Now due to kill(), kernel sends "SIGINT" signed to the process. Though kernel sent it, it does not notice about the signal here. Rather as explained before it checks about the signal when it returns to user mode.

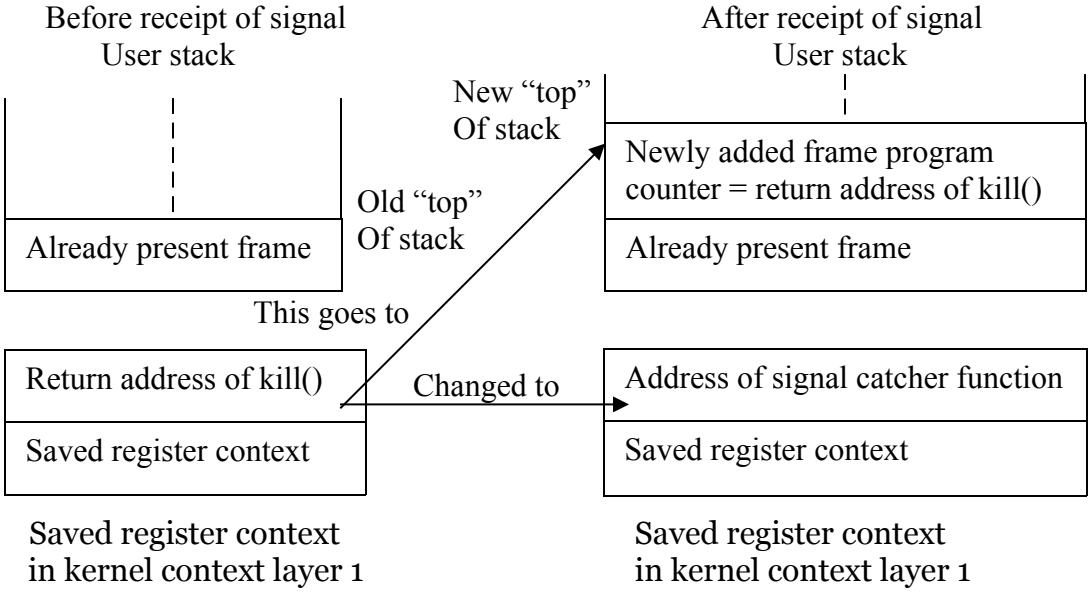
- **Here the 5 steps occurs sequentially…**
    - ⇨ Kernel first accesses the "SAVED REGISTER CONTEXT" of the process and finds out program counter and stack pointer.
    - ⇨ Then from the process's U area, it clears the entry of catcher() function from the array of "SIGNAL NUMBER – SIGNAL CATCHER FUNCTION".
    - ⇨ Then at this "CLEARED FIELD" it sets 0 as default action for this signal.

- ⇨ Then it creates a "NEW STACK FRAME" over the existing stack frame (which was pushed when process enters in kernel mode due to call to kill) and inside that new frame it writes...

➔ Retrieved program counter (taken from step (a)).
➔ Retrieved stack pointer (taken from step(a)).
➔ Return address of kill() i.e. AddOfreturnOfKill.

So here kernel pretends as if it already called kill() system call.

- ⇨ Then it goes to "SAVED REGISTER CONTEXT" again. Here program counter already has "RETURN ADDRESS" of kill() means AddOfreturnOfKill. It wipes it off and writes address of signal catcher function in place of it. So kernel writes AddOfCatcher in place of AddOfReturnOfKill.
  It also increments the stack pointer to print to the newly added stack frame.



<u>Anomalies In psig() Algorithm</u> :- There are several anomalies in psig() algorithm of handling signals.

[1] <u>Clearing the field in U area which contains the address of signal catcher function</u>: - As mentioned in algorithm, kernel clears the field in U area which actually has address of "SIGNAL CATCHER FUNCTION". This statement has un-desirable effects.

This is done to allow the process to call signal() system call again when the process wants to handle the same signal. Though this is the required thing it has a major race condition because...
(a) Second instance of the signal may arrive before the process has a chance to execute the system call.
(b) As process here is now in "USER MODE", kernel can make a context switch and the "CHANCE OF RECEIVING THE SIGNAL BEFORE RESETING THE SIGNAL CATCHER FUNCTION", increases.

The following program states this race condition.

```
# include <signal.h>
void signal_catcher(void)
void main(void)
{
        /* Variable */
        int ppid;
        /* Code */
```

```
        signal(SIGINT, signal_catcher);
        if(fork() == 0)
        {
         /* Give enough time to both processes to set up by using
         sleep() */
         sleep(5); /* library function for 5ms delay */
         ppid = getppid(); /* Get parent process id */
         for(;;)
           {
               if(kill(ppid, SIGINT) == -1)  exit(0);
           }
        }
         /* Lower the priority to increase chances of race */
         nice(10);
         for(;;)
}
```

➔ First process calls signal() system call to catch the "INTERRUPT SIGNAL" and then allow to execute the signal catcher function, the signal catcher function is....

```
Void signal_catcher(void)
{
         /* Code */
         printf("PID of %d caught one \n", getpid());
       /* This prints process ID */
         signal(SIGINT, signal_catcher);
}
```

➔ Then process creates a child process by using fork() system. The following code inside the "if statement" of fork() is only for child process.

➔ The last 2 lines of code are again only for parent process. In this code, the parent process "LOWERS' its priority by using nice() system call. The value 10 used as its parameter, is going to be used in priority calculating equation and called as "NICE VALUE". Thus this nice() system call lowers priority of the parent process as compared to its child.

➔ Then parent process enters in an infinite loop.

➔ Now see the child's code inside "IF BLOCK" of fork() ……..

➔ The child process suspends its execution for 5 seconds to give the parent process a time to execute nice() call to lower its own (i.e. parent) priority.

➔ Now child gets parent process's id by using getpid() call.

➔ Then child enters loop and sends "INTERRUPT SIGNAL" to its parent by using kill() system call with parent process ID, during each iteration. If due to some reason kill() fails (most probably if parent does not exist), the kill() will return –1 and child will exit.

➔ The idea behind this program is that when child sends **SIGINT** to its parent, as parent is existing (due to infinite loop) and as parent sets **SIGINT** by signal() system call with signal catcher function, hence the parent will execute its signal catcher function every time when it receives **SIGINT** signal.

➔ When parent receives **SIGINT** signal, it executes the signal catcher function " SIGNAL_CATCHER() ", in which it prints a message (i.e. its id) and calls signal() system call again to catch the next occurrence of same signal.

➔ And continues its execution due to infinite loop.

Possibly following events occur sequentially –

1. Child sends interrupt signal to parent.
2. Parent catches the signal, calls signal catcher function but kernel may preempt the parent (as parent is now in user mode) and can make context switch before it executes the signal() system call written inside its signal catcher function.
3. Though parent gets preempted, its child is independent and thus can send same interrupt signal again & again to its parent.
4. parent gets rescheduled, receives the message again but as it yet not called signal() in its signal catcher function, there is no arrangement of parent for this new arrival of signal, and as up till now kernel erased the name of signal catcher function from parent's U area and replaced it by 0 (default exit action), parent exits without handling the signal.
   ➔ This program is deliberately written to show such type of race condition. Because deliberate call to nice() lowers the priority of parent than its child and thus child is executed (rather scheduled) more times than parent and hence above race occurs. In practice it is not possible to tell exactly when this race occurs.

According to ***Dennis Ritchie***, at the beginning period of UNIX, signals were designed as events for fatal conditions or ignorable conditions of the process.
That is why these signal were not for the "PROCESS HANDLING" purpose. Means at that time signal were either allow default action of exit or allow to be ignored by the process. They were not meant for handling. Thus above race conditions are also ignored.
But later, as the development of UNIX progressed, now users need to handle the signals.

- ! This problem had one solution at that time. Means do not allow kernel to clear the "SIGNAL FIELD" & the "SIGNAL CATCHER FUNCTION" field in the process table & the U area array respectively. But this leads to another problem. Means if the signals are allowed to keep on arriving (due to not to clear above fields) and process is allowed to catch it and then respond to it with signal catcher function, then every evoke of "SIGNAL CATCHER FUNCTION", its stack will be pushed into the user stack of the process & Thus this stack may grow out of bounds.
- ! Another solution was that, kernel will reset the value of signal handling function in U area array to 1 to ignore such type of signals, until process again specifies what to do with these signals. But this creates another problem that, as signals are ignored, there is no way for the process to know how many signals it received. But this is less severe than before.
- ! Finally BSD system allows a process to block or unblock the receipt of signal with creating of new type of signal call in their system. By using this new system call, now process unblocks a signal and thus immediately kernel sends the pending signals, which were blocked. When process receives a signal, the kernel automatically blocks further receipt of the signal until the signal catcher function completes its execution.Thus now signal handling becomes analogous to interrupt handling too, kernel blocks further receipt of interrupts until it completes with the current interrupt handler function.

[2] Catching of signals those occur while process is in a system call or process is in "INTERRUPTIBLE SLEEP" :- Such signal causes process to take longjmp(as seen in sleep() algorithm) algorithm to come out of sleep, then come to user mode and then call the signal catcher function.

So when process returns from signal catcher function, it looks like process returns from system call, with an error indicating that the system call was interrupt.

So user can then check the error and re invoke the system call to avoid this anomaly.

But BSD gives better solution. It forces the kernel to restart the system call when above anomaly occurs.

[3] Ignoring the signal by process :- This anomaly is mainly seen when process's sleep is set to "INTERRUPTIBLE BY SIGNAL" priority but process will not do longjmp algorithm during coming out of sleep.(i.e. return (1) condition in sleep algorithm).

So kernel realizes that process had ignored the signal. But this will occur only when process wakes up and starts running. So kernel gets no time to know when to start signal catcher function.

One solution was there to keep the process in sleep. But sometimes process's U area will not be accessible to kernel to know the U area because signal catcher function's address is stored in "U AREA ARRAY".

Another solution was there that to avoid the un-accessibility of U area, store the address of signal catcher function in process table. As it is global kernel will have "ALL TIME" access to it. Now kernel can decide whether to awake the process on receipt of signal or not.

But sometimes process may again go to sleep in sleep algorithm itself when it found that it is not awakened yet. As seen in chapter 2, kernel encloses re-entry to sleep algorithm in "WHILE LOOP", putting the process back in sleep if event did not occur really.

[4] Not treating "DEATH OF CHILD" signal as equally as other signals :- When process recognizes that it received "DEATH OF CHILD" signal, it turns "OFF" the signal bit in process table entry. In "DEFAULT" case it acts as if no signal is arrived. (see issig() algorithm). If process is slept in "INTERRUPTIBLE" priority and if "DEATH OF CHILD" is received, then process wakes up. Because the sleep algorithm doesn't check for type of signal. If process decides to catch "DEATH OF CHILD" signal, then it can invoke its signal catcher function.(death of child)

If "DEATH OF CHILD" is ignored then case is different than other signals. We will see about it in wait() system call.

But **Note that,** if process wants to catch "DEATH OF CHILD" signal by using signal() system call, the kernel will send "DEATH OF CHILD" signal only when the process has a child in zombie.

Process Groups: - Though its unique PID number identifies process, sometimes it is must to identify the process by its "GROUP" besides the PID.

E.g.: - When processes have common ancestor process, such as "SHELL", and then these processes are said to be related by a "GROUP". In such cases, when user presses "DELETE" or "BREAK" keys on terminal's keyboard or when terminal hangs up, then such signal is sent to all these single "GROUP" related processes.

Kernel uses process group ID to identify the "MUTUALLY RELATED PROCESS". These related process, which have common "GROUP ID" should receive common signals for certain events.

Kernel saves this group Id in process table (see contents of process table (5^{th} content)) entry and again note that processes in "SAME PROCESS GROUP" have identified group ID.

The system call for process group ID is the setpgrp() . (Where "P" for process, "GRP" for group) when it is called in a process, then this system call gives process group id equal to the caller process's PID. The syntax is .....

grp = setpgrp();

Where grp is the new "PROCESS GROUP ID".

* child of a process always inherits its parents "PROCESS GROUP ID" as its own "PROCESS GROUP ID", during the fork() system call.

How Process To Process Signaling Occurs ?

Process use kill() system call to send signals to each other. The syntax is...

Kill(pid, signal_number);

Where pid is the process's id to whom this process wants to send the signal. And signal_number is the number of signal that process wants to send.

The parameter pid has different meanings which are as follows…

I. If pid is +ve integer, kernel send the respective signal to the process having its process ID as pid.
II. If pid is 0, then kernel sends respective signal to all processes in the sender's process's group.
III. If pid is –1, then kernel sends respective signal to all processes whose "REAL USER ID" is equal to the "EFFECTIVE USER ID" of the sender. Here if the sending process has its "EFFECTIVE USER ID" equal to the "EFFECTIVE USER ID" of the super user, then kernel sends respective signal to all processes except process 0 (swapper) & process 1 (init) .
IV. If pid is –ve value, but not –1, then kernel sends respective signal to all processes in that group which is equal to the absolute value of pid.

Note that → in all cases…

1. If the sending process does not have "EFFECTIVE USER ID" equal to the "EFFECTIVE USER ID" of super user or
2. If the sending process "REAL USER ID" or "EFFECTIVE USER ID" does not match with the "REAL USER ID" or "EFFECTIVE USER ID" of the "RECEIVING PROCESS", the call to the kill() system call fails and as usual returns –1.

Following example shows the use of the setpgrp() & kill() together …

Example: -

```
#include<signal.h>
void main(void)
{
        /* variable declarations*/
        register i;
        /* code*/
        setpgrp();
        for(i=0; i<10; i++)
        {
                if(fork() == 0)
                {
                        /* code of child only */
                        if(i & 1)
                                setpgrp();
                        printf ("pid = %d pgrp = %d \n", getpid(), getpgrp());
                        pause (); /* suspend execution */
                 }
         }
         kill(0, SIGINT);
}
```

- In above program setpgrp() call resets the process group ID.
- Then it creates 10-child process.
- Each child process has the same "PROCESS GROUP ID" as that of its parent.
- But **Note that** when the created process is in "ODD ITERATION"(due to i & 1) of the loop, then it resets process group ID, to the executing process at that time.
- The pause() system call suspends the currently executing process to allow to parent to call kill) so that child will receive the signal (due to first 0 parameter)

- The kill() system call by parent will go to all processes in its process group(means to all "EVEN" iterated processes, because they have not reset their process group Id hence their process group Id is that of parents process group ID).

  But "ODD" iterated processes will not receive signals because they reset their process group ID and hence parent's process group ID & their process group ID's become different.

Process Termination: - Processes in UNIX get terminated by executing the exit() system call.

The exiting process first enters in zombie state (as shown in process transition diagram) before actually exit. In zombie state the process "GIVE UP" it's all occupied resources, then dismantles its whole context but leaves its slot in process table occupied.

The syntax for exit() system call is ....

Exit(status);

Where the "STATUS" is returned to the parent process by this process as "EXIT STATUS OF CHILD" (remember that every process in UNIX except process 0 is child of some parent process). So parent can use this "STATUS" for examination of child's exit.

- Any process can call exit implicitly or explicitly "IMPICITLY" means, if child does not call exit() as part of its code, it gets exit after termination of main() function of it. (**Note that** – main() function is an entry point function of every process in UNIX). "EXPLICITLY" means a process can call exit() as part of its code anywhere in its body. When this gets executed the process exits even some code is there after this line of exit().
- Kernel itself can invoke exit() system call internally when a process receives an uncaught signal. (this is default action of kernel for uncaught signals as explained in previous topic of signals). In this case the "STATUS" (i.e. parameter, which is going to be return value to parent for exit()) is the signal's number.

Note that, system has not limit for "EXECUTION TIME" of a process. So processes may live long without encountering exit() call.

E g: -

1. Process 0 (i.e. swapper process) & process 1 (i.e. init) live throughout the lifetime of the system and they only exit when system is shutdown or rebooted.
2. Another example of such "LONG LIVING" process is getty(), which is a "TERMINAL MONITOR" process and hence remains alive uptill the terminal is switched off.

The algorithm for exit() system call is as follows .....

```
01: Algorithm: exit()
02: Input: return code / exit status going to parent process
03: {
04:   ignore all signals;
05:   if (this process is "process group leader" having control terminal)
06:   {
07:         send "hang up signal" to all processes of this same process group;
08:         reset new process group for all remaining processes of this group as 0;
09:   }
10:   close all open files using special version of close;
11:   release "current directory inode" by iput;
12:   release "current root inode" by iput;
13:   free regions, page table associated with this exiting process by using detachreg;
14:   write accounting records to global "account" file;
15:   make process state zombie;
```

16: if any children of this process are there, then assign "parent process ID" as of init process (i.e. 1);
17: if any children are in zombie state, send "death of child" signal to init (as init is now parent of all children);
18: send its own "death of child" (for its own death) to its parent;
19; do context switch on its own;
20: }
21: output: none

- Kernel first ignores all signals coming to this process because as this process is going to exit(), there is no sense to keep on catching signals.
- If sometimes in past, this process had called setpgrp() to become leader of its children and suppose this process also has terminal's control, then .....
    a) Send "HANG UP" signal to all processes of this process's group.
    b) As this process is going to exit, its children will require new "GROUP LEADER". Hence kernel sets group ID as 0 for all these processes of this process's group. This is done in anticipation that some other process may call setpgrp() to become leader of this group. Or some other process may get pid equal to this process's pid to become leader of this group. Right kernel made group Id as 0 means all these process belong to group of process 0 **Note that** the process belong to "OLD" process group will not belong to the "LATER" process group.
- Then kernel close all open files of this process, one-by-one by using internal variation of close() system call.
- Then kernel releases "CURRENT DIRECTORY" inode by using ipnt algorithm.
- Similarly kernel also releases "CURRENT ROOT" inode by using ipnt algorithm
- Then kernel releases all memory means regions, page tables etc by using detachreg() algorithm. But it saves
    1. Exit status code
    2. Accumulated user time
    3. Kernel execution time for this process & for its children in the "PROCESS TABLE SLOT" of this process.
- Kernel also writes an accounting record of this process to a global accounting file . This accounting record contains
    1. User ID of process
    2. CPU usage
    3. Memory usage
    4. Amount of I'O of the process.
    5. pid [process id]
- Some application program to get "PROCESS STATISTIC" of the system, which is mainly required in system's "PERFORMANCE MONITORIZATION" & customer billing, can read this global "ACCOUNTING FILE" later.
- Now kernel makes process's state "ZOMBIE" & thus makes relevant change in "STATE" field of this process's process table slot.
- As this process is now going to exit, its children will loose their parent. So kernel makes "INIT" process as new parent of these child processes. To do this kernel assigns id of init to all these children as their "PARENT PROCESS ID". This id is 1. This task of kernel detaches this exiting process from system's process tree.
- Sometimes it may happen that when this process is exiting, any of its children may be already in zombie state, actually this is to be informed to the parent of this child. But as parent is going to exit and thus "INIT" is now new parent, this exiting process tells init (new parent) about its zombie children by sending "DEATH OF CHILD" signal. So that later "INIT" can remove zombie child's process table slot.

- Now this exiting process also has its own parent (not the init but the creator of this process) and as this process is now exiting, it itself sends "DEATH OF CHILD" signal to its own parent.
- Now this "ZOMBIE STATE" process forces kernel to make "CONTEXT SWITCH" so that kernel can schedule other process.

Note that: - Kernel never schedule zombie process to execute because zombie process is surely going to exit soon.

Example: - following is the example of exit.

```
void main(void)
{
        /* variable declaration */
        int child;
        /* code */
        if((child = fork()) = = 0)
        {
                /* child's code */
                printf("child PID = %d\n",getpid());
                pause(); /* suspend execution until signal*/
          }
          /* parent's code */
          printf("child pid = %d \n ",child);
          exit(child);
}
```

- Here this process creates a child process which prints its pid by using getpid().
- Then child calls pause() system call, suspending execution of child for some time until it receives the signal.
- Due to pause(), when parent gets scheduled, it will print child's pid by using "CHILD" variable and then parent exits, returning child's pid as its exit status code.

  Actually there was no need of this explicit exit(), because after main's termination exit() is must to occur.
- Here child will go on (though paused until signal) though its parent does not exist.

* Waiting For Process Termination: -

Many times a process wants to synchronize its execution with the termination (i.e. exit) of its child process.

For this purpose wait() system call is used, whose syntax is .....

pid = wait(status_address);

where pid is the process ID of that zombie child with whose termination the process wants to synchronize. And status_address is the address of an integer where "EXIT STATUS CODE" of the child, will be stored.

The algorithm of wait() system call is as follows .....

```
Algorithm: wait()
Input: address of integer variable to store status
         of exiting process
{
        if(waiting process has no child process)
                return(error);
```

```
        for(; ;)
         {
           if(waiting process has zombie child)
              {
                      pick arbitrary zombie child;
                      add child's CPU usage to parent;
                      free child process's process table entry;
                      return(child's ID & child's exit status);
              }
           if(process has no children)
              return(error);
           sleep at "interruptible priority" (event : child process exits);
        }
}
Output: child's process id as return value & child's exit status as parameterized variable
```

- Kernel status searching of zombie child of this waiting process and if there is no child process found, returns error.
- But if it finds a zombie child, it enters in an infinite "FOR LOOP" and then .....
  - Picks up this zombie child process.
  - Adds "ACCUMULATED TIME OF CHILD PROCESS"(both in user mode and kernel mode) i.e. the CPU usage, to the appropriate field of parent's U area.
  - Releases this zombie child's process table entry. This slot is now available for any other new process.
  - Finally returns pid number of zombie child & exit status of the zombie child together to the parent process of this zombie child.
- Then it again checks for any child of this process in this infinite loop & returns error when no child remains.
- But when process has a child but not in "ZOMBIE STATE", then process sleeps at "INTERRUPTIBLE" priority until the event of "ANY ZOMBIE CHILD" occurs.
- The loop terminates only when internal statements of return is executed, otherwise this is an infinite loop.

Some important explanation: - When a process has a child but not in zombie state, as explained before process sleeps in "INTERRUPTIBLE" priority until at least one child goes in zombie state.(i.e. at least one child exits).

As explained in the topic of "SIGNALS", when a process sleeps at "INTERRUPTIBLE" priority, then it can wake up only when it receives signal. Because kernel has no explicit wake up call in wait() algorithm for this situation.

So for any type of signal the process will wake up on receipt of signal.

But for the signal "DEATH OF CHILD", the process in wait() may respond .....

(a) By default: - When signal i.e. "DEATH OF CHILD" arrives, process wakes up from its sleep in wait() algorithm. Then sleep invokes issig() algorithm (though in sleep() algorithm nothing is written about issig(), it is mentioned that algorithm searches for any "PENDING SIGNAL" . this search is nothing but issig() algorithm), to check any existence of pending signal. Then issig() recognizes special situation of "DEATH OF CHILD" signal and returns false(see issig()). Therefore in wait() concerned sleep, kernel does not make longjmp i.e. return(1) sleep() from sleep and lust returns to wait().

Kernel again restarts the loop in wait() and finds for any zombie child, but uptill now "AT LEAST ONE CHILD" is in zombie state (because "DEATH OF CHILD" signal sent previously indicates that), and hence proceeds through the "IF BLOCK" of valid zombie child as usual.

(b) If catching of "DEATH OF CHILD" is decided: - In such cases process wakes up from its sleep in wait() and returns to user mode to handle the signal catcher function which was set by signal() system call by this process. So this behavior of process for "DEATH OF CHILD" is just like for other signals.

(c) If "DEATH OF CHILD" signal is decide to be ignored: - Kernel restarts the wait's for loop again enters into the "IF BLOCK" of zombie children and again starts searching for new zombie child. As "NO CHILDREN" occurs, it returns from wait(1) loop, with error. Here while searching for pending signals in sleep(), issig() fails, thus in sleep() longjmp is done. Kernel returns back in wait(), but up till now "SIGNAL SIG_ID" is cleared, hence it starts loop again. When all children exit, it returns from wait() with error as "NO CHLDREN".

Example program 1: - This example is of using wait() system call and ignoring "DEATH OF CHILD" signal.

```
# include <signal.h>
void main(int argc, char *argv[])
{
        /* variable declarations */
        int i, ret_val, ret_code;
        /* code */
        if(argc > 1)
                signal(SIGCLD, SIG_IGN); /* ignore "death of child" signal */
        for(i = 0; i < 15; i++)
        {
                if(fork() ==0)
                {
                        /* code of child only*/
                        printf("child's process ID = %d\n", getpid());
                        exit(i);
                }
        }
        /* parent code */
        ret_val = wait (& ret_code);
        printf("wait ret_val = %x AND ret_code = %x\n", ret_val, ret_code);
}
```

The output of the above program is different for "USING NO COMMAND LINE ARGUMENTS" & "USING COMMAND LINE ARGUMENTS".

(1) Suppose this program is executed without using command line arguments: - Means there will be only "PROGRAM'S NAME" on command line (means argc == 1).

In this case parent creates 15 children which exit with "EXIT STATUS CODE" as i.

When parent code is executed, wait() will be executed, which will find zombie child & returns from wait. But it is not sure which zombie child (out of 15) it will find.

Some explanation is necessary for exit(i) :- The "**C**" library code for exit () stores the actual "EXIT CODE"(i.e. exit status) in 8 to 15 bits of ret_code (remember in c "INT" is of 2 bytes i.e. 16 bits out of

which 8 to 15 are used for exit status) this ret_code equals 256 * i (because 8n to 15 bits means total 8 bits & 2^8 =256) depending upon value of i. The ret_val equals the value of child process ID.

(2) Suppose this program is executed with command line arguments: - Then argc will be > 1 and thus signal () call will be executed. In this signal call the signal number of "DEATH OF CHILD" is specified by constant SIGCLD and the ignoring constant is SIG-IGN. This line tells us that parent wants to be informed for "DEATH OF CHILD" signal, though parent is going to be ignore it.

Assume that parent sleeps in wait() before any of its child exit, then actually child exits, kernel will send "DEATH OF CHILD" signal to the parent process and thus parent will wake up as its sleep is set for "INTERRUPTIBLE" priority for signals. So parent wakes up, kernel schedules it to run and when parent arrives in user mode, it finds that it has a "DEATH OF CHILD" signal to process but as it is going to ignore it, kernel removes the entry for zombie child from process table (see issig ()) and continues executing wait() system call as if nothing is happened. This same procedure will occur every time parent receives "DEATH OF CHILD" signal. Ultimately all children will exit and in wait() algorithm condition of "NO ZOMBIE CHILD" will occur due to which wait() will return with error, -1.

The difference between (1) & (2) is that, the parent waits for the termination of any child in (1), but waits for the termination of all children in (2).

Note that: - Older version of UNIX implemented exit() & wait() system call without considering "DEATH OF CHILD" signal.

Example program 2 ➔ This example shows the necessity of including "DEATH OF CHILD" signal in exit() & wait() system calls, which was not done in older versions of UNIX.

```
#include <signal.h>
void main(int argc, char *argv[])
{
        /* variable declaration */
        char buffer[256];
        /*code*/
        if (argc != 1)
                signal(SIGCLD, SIG-IGN);
        while(read (0, buffer,256))
        {
                if(fork() = = 0)
                {
                        /* child code */
                        printf ("%s \n",buffer);
                        exit(0);
                }
        }
}
```

Suppose "DEATH OF CILD" is not considered in exit() :- Here parent process reads from its standard input (i.e. 0) until "END OF FILE" is not specified by user.

But as parent does not wait for implementation of child's code and goes on sending read data to created children (because wait () is not used in parent).

Now as signal () call is used, it will clear the entries of zombie child process automatically from the process table.

But think if signal() is not used, then the entries of the zombie children will remain in process table ultimately filling the process table up to its max capacity and crashing the system.

That is why, though ignored, consideration for "DEATH OF CHILD" is must.

This is another reason why wait() is used in parent because if wait is not used, parent will not sleep and will not call signal to clear the process table entry of zombie children when child actually exits.

Invoking Other Programs: - exec() The exec() system call invokes other programs when a process calls it. While doing this the memory space of calling process is overlaid with an executable image of an executable file (i.e. the program to be invoked)

Because of this the original contents of "USER LEVEL CONTEXT" of the calling process become un-accessible except the parameters of exec() system call. This is because during overlaying process kernel copies these parameters from process's address space to some other address space in memory.

The syntax of exec() system call is ....

execve (filename, argv, envp);

where filename is the name of the executable program's file which is to be invoked. Argv is pointer to an array of character pointer which are strings for the parameters to executable program (if any) and envp is another pointer to an array of a character pointers which are strings for "ENVIRONMENT" of the executable program.

There are several library flavors of exec() system call like execl, execv, execle etc. but all of them ultimately call execve(). Thus is used here as representative of exec() system call.

As we know a process in UNIX has main() function. When process has command line arguments, the main function looks like main(argc, argv). Here the argv of main is the copy of argv to exec(). The characters strings in envp array are of the form "name = value", which contains some useful information for executable program like user's home directory, path of directories to search for executable program etc. processes can access their environment via the global variable environ. This variable is initialized by "**C**" startup routines.

The algorithm for exec() system call is .....

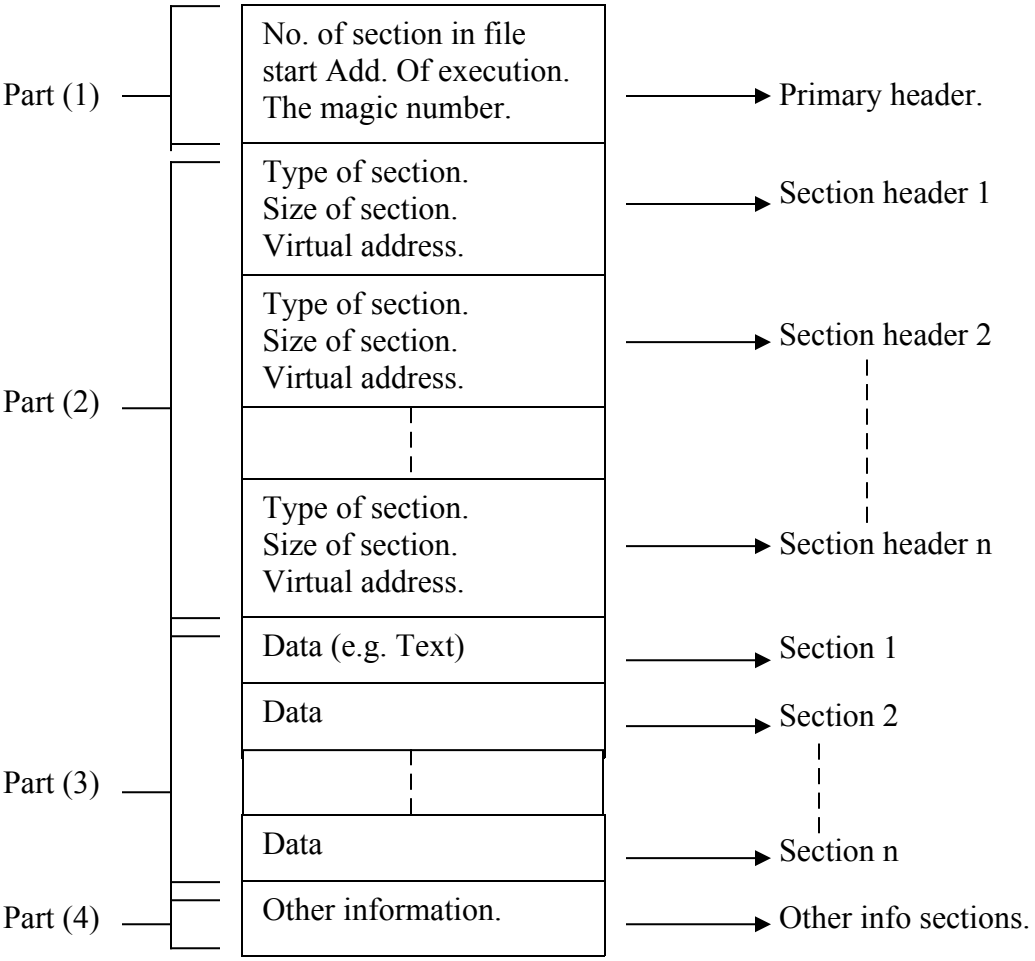
```
01:  algorithm: exec()
02:  input: 1. executable file's name (with path).
03:         2. parameter list to executable program.
04:         3. environment variable list.
05 : {
06:    get executable file's inode by namei();
07:    verify that the file is really executable and also verify that user has executable permission for it;
08:    read executable file's headers and check whether this executable file is "loadable module";
09:    copy parameter list of exec() from old address space to system's address space ;
10:    for(every region attached to the process)
11:          detach all old regions using detachreg();
12:    for (every region in executable files "load module")
13:    {
14:          allocate new region by using allocreg() ;
15:          attach these new regions to process by attachreg;
16:          load regions into memory by loadreg;
17:    }
18:     copy "previously copied" exec's parameter list from system's address space to newly created
        user stack region;
19:    do special processing of setuid() for executable program & trace;
20:    initialize "user register save area" for return back to user mode;
21:    release inode of executable file by ipnt();
22: }
```

23: output: nothing

- First kernel gets the executable file's inode by using namei() algorithm for the path specified in 1st parameter.
- Then it determines the file is really "EXECUTABLE" or not by studying its inode. It also determines whether the user has "EXECUTABLE" permission for executing this file.
- Kernel then reads executable file's header to study its layout. This layout is generated either by assembler or loader.

Layout of an executable file :-

The diagram of layout of an executable is as follows .....



The logical layout of the executable file is made up of 4 parts.

1. Primary header
2. Section headers
3. Sections
4. Miscellaneous sections

(1) Primary header: - This is only one in number. It contains information about

- How many sections are in executable file
- The starting address of process execution (i.e. initial register value)
- The magic number, which gives the type of executable file.

(2) Sections header: - This describes each section in the executable file. Thus there numbers depend upon number of sections in the executable file (say 1 to n). It contains information about

- Type of section.
- Size of section
- Virtual address that this section should occupy when running in the system
- Other information about section.

(3) Actual sections: - This is the actual section part (above part was information about these sections). This contains information about the "DATA" in this section, which is going to be loaded in the process's address space. The data may be the text, actual data or stack etc. the number of these sections is variable according to the information in the executable file (say 1 to n).

(4) Miscellaneous sections :- These sections may contain information about symbol table and other data useful for debugging.

* The above format is still evolving as technology advances, but one thing is common & sure that every executable file contains one primary header with Magic number.

The magic number is a short integer, which identifies this executable file as "LOADABLE MODULE" and enables kernel to distinguish "RUN TIME CHARACTERISTICS" of this loadable module. Magic number also plays an important role in paging system of system's memory management.

- Now coming back to algorithm of exec, by verifying layout of executable file kernel determines that this file is a "LOADABLE MODULE". As this image of executable file is going to "OVERLAY" on existing address space of calling process, kernel first must free the current address space of the process.

  But problem right now is that this address space also contains parameters to the exec() system call in user stack.

  So kernel first copies these parameters of exec() from this address space to a temporary buffer in system's memory. While copying it finds that these are arrays (i.e. argv[], envp[]) of character strings, thus it cannot just copy these addresses. So first it copies individual character strings, and then copies individual character strings. The most common place of copying is to the kernel's stack. Alternatively it may copy these to an unallocated area in memory (borrowed temporarily) or to a secondary storage device such as swapping device.

  Though the commonest place is kernel's stack, the kernel stack has a bounding limit(recall that to keep system design simpler, most data structures in kernel are always and arrays always have some limit) and as parameters of exec() are "VARIABLE SIZED", choosing "KERNEL STACK" only may be risky. Thus it may choose "UNALLOCATED MEMORY WHICH IS TEMPORARILY BORROWED" (faster) or secondary storage device (comparatively slower). Out of these choices kernel always chose the possible "SIMPLER AND FASTER" at that time.
- Now kernel is ready to free the current address space of the calling process. Hence it uses detachreg() algorithm for every region in the process's current address space.

  Now onwards the process has "NO ADDRESS SPACE" and hence if any errors occur result in "SIGNAL" which will terminate the process. Such errors may include
  - ⇨ Running out of space in kernel's global region table or
  - ⇨ Attempt to load such a program whose size is out of the limit given by the system.
  - ⇨ Attempt to load such a program whose region address is overlapping the other regions.

  If no errors occur in above step, kernel now allocates new region by using allocreg(), then attaches these new regions to the process by using attachreg() and loads them by using loadreg().

  While doing this, right now kernel works only for text & data regions (stack region? not yet).

  Initially the data region (newly created) has 2 parts
  - ✓ Data initialized at compile time
  - ✓ Data not initialized at compile time (bss).

So during allocation & attachment of data region, first initialized data is considered. Then kernel increases the size of data region by growreg() for the un-initialized data(bss), and initialize these values to 0.

Now it works for stack region. It allocates new region for stack (allocreg()), attaches it to the process(attachreg()) and most importantly it allocates memory for exec's parameters which are now going to be the part of this stack.

- Then kernel works for exec's parameters. If these parameters are previously copied to the "UNALLOCATED MEMORY PAGES", then kernel can use these pages directly as a part of this stack (recall that stack is contiguous but pages may not be contiguous). But if these parameters were copied to kernel's stack or to secondary storage device, then kernel has to copy them back to this stack.
- Though not written in algorithm, kernel clears the address of signal catcher functions in process's "U AREA" because as now process's address space is newly created, these "PAST ADDRESS" are meaningless. Also **Note that** ignorable signals remain ignorable in this new context too.
- Kernel takes special precautions for setuid() and for process tracing. (will be seen later).
- Now kernel goes to "REGISTER CONTEXT" of the process. This context is going to be used by process when it will run in user mode. Here kernel specifically sets stack Pointer to new grown stack's top and then sets the program counter to the starting address of execution given by the ("PRIMARY HEADER" of the executable file).
- Finally kernel invokes iput algorithm to release the inode of the executable file which was locked in first step of this algorithm in namei(). **Note that,** as it is executable file it cannot be really opened or closed by open() & close() algorithm. But accessing its inode in namei() & releasing its inode in iput() is similar to above opening & closing of regular file. The difference is that such direct opening and closing of executable file (i.e. by namei() & iput()) will not have any file table entry nor it will have file descriptor. But surely it will have "INODE TABLE" entry.

Here ends the exec() algorithm. But some points are important to remember .....

- ✓ When process returns from exec() call, it starts executing of new program which overlaid on it.
- ✓ Though overlayed by new program, this is the same process as before exec(), hence its process ID will not change, nor its place in process hierarchy will change. The change only is in "USER LEVEL CONTEXT".

Example: - Following code is an example of exec

```
void main(void)
{
        /*variable declaration */
        int status;
        /* code*/
        if(fork() = = 0)
        {
                /* child's code */
                execl("/bin/date","date",0);
        }
        /*parent's code */
        wait (& status);
}
```

Above program process has parent process, which calls fork() to create child process. Here child process then calls execl() system call to execute "DATE" program, and parent waits until child's exit.

So here it is very important to **Note that** as soon as fork() completes, parent & child has different address spaces. So the program "DATE" is going to overlay child's address space (because child called it) and not that of the parent's address space.

Before invoking the actual exec(), child right now has "INSTRUCTIONS FOR IT" (not for "DATE") in its text region, then its data region consist of strings "/BIN/DATE" & "DATE" and its stack region consists of the frame to get the exec call.

Now kernel finds "/BIN/DATE" file by names(), verifies that it is executable and user has "EXECUTE" permission.

Conventionally first parameter is pathname of the executable program i.e. argv[0] here.

Kernel then copies the string "/BIN/DATE" & "DATE" to some memory locations and then frees text, data and stack regions of child.

Then it allocates new text, data and stack regions, copies "INSTRUCTION SEXTION" (i.e. text of "DATE") of "DATE" program to the text region of child, then copies "DATA SECTION" of "DATE" to data region of child.

Kernel now creates stack and copies back the saved (previously copied to some memory) parameters to this new stack.

So after call to exec, the child will not execute any of its previous code (though this example does not have any other code besides exec in child), but will execute the "DATE" program.

After completion of date program, child will exit, and parent receives child's "EXIT STATUS" from the wait() system call through "STATUS" variable (which was empty when passed and gets filled with child's exit status when child exits).

Up till now we considered Text region & data region as two separate regions. And this is correct too.

But in past, earlier versions of UNIX allowed text & data regions to be in the same region (combined) this was done because the machine on which UNIX developed was PDP, which has much limitation on the size of process. Programs at that time were smaller& require less "SEGMENT REGISTERS" if text & data regions are kept combined.

But later versions of UNIX separate these two regions due to some more advantages & to avoid some disasters.

The 2 major advantages of keeping text & data regions as separate regions are…

(1) Protection &
(2) Sharing.

(1) <u>Protection</u>: - If text & data regions are kept same, then system cannot prevent a process from overwriting its instructions (i.e. text region), because if these 2 regions are same, system can not able to know which address contain instructions & which address contain data.

But if they are kept separate, system can set up some hardware protection mechanism to prevent other processes from overwriting their regions. Means if a process mistakenly tries to overwrite it text region's address space, kernel can issue "GENERAL PROTECTION FAULT" which will ultimately result in a signal which will ultimately terminate the process by kernel's exit() call for such process.

To understand above thing, let us take one example program …..

```
#include <signal.h>
void signal_catcher(int);
void main(void)
{
 /* variable declarations */
```

```
int i, *ip;
/* function declarations */
void function (void);
/* code*/
 ip = (int *) function; / assign address of function to ip*/

for (i = 0; i< 20; i++) /* as there are 19 signals known*/
        signal(i, signal_catcher);
*ip  = 1; /* attempt to overwrite address of function*/
printf("this is the line after assignment to ip \n");
function(); /* call to function*/
}
void signal_catcher(int n)
{
/*code*/
printf("the catched signal is %d \n",n);
exit(1); /*with some error*/
}
void function (void)
{
/* body of function */
}
```

- This program first assign address of a function, function() to ip pointer. Then it makes arrangement to catch signals.
- Then it tries to overwrite the address in ip by some other address (the integer 1 will have its own address).

  Now if we consider that text & data regions of this process are separate, then as soon as overwriting of ip occurs, kernel gives "PROTECTION FAULT" because Function() is "TEXT PART" of process and this statement is trying to overwrite its address (i.e. one of the address in text region which points to the function()) which is in "WRITE-PROTECTED" text region.

  So immediately kernel sends SIGBUS signal (i.e. busy) to the process which catches the signal and executes signal_catcher() and exits by printing statement in signal_catcher() but not printing the printf() in main().

  But if we consider that text & data regions are combined in a signal region, the kernel will not realize that it is overwriting the address of function(). So address of function will become &1 and printf() in main() will be executed. But when the flow arrives at the call to function() it will execute an illegal instruction "&1" and kernel will send SIGILL (i.e. ILLEGAL) signal and then process will execute signal_catcher() which will print a line and then exit the process.

(2) <u>Sharing</u>: - The second advantage of keeping text & data regions as separate regions, is to allow sharing of regions between different processes.

As we are keeping text & data regions separate & we are also keeping "TEXT REGION" as "WRITE PROTECTED", the "TEXT REGION" is never going to change at all from the time when kernel loads it into the memory copying from an executable file.

So in memory this "TEXT REGION" is going to remain un-touched. Now if more than one process wants to call the same program whose "TEXT REGION" is untouchably loaded into the memory, then they can share this text region to execute the code of that program. This will also save much memory. Because if sharing is not allowed every process, when want to call a program should load separate copy of "TEXT REGION" of that program which will occupy much memory.

Thus during exec(), when kernel allocates a text region for a process, it checks whether the executable program allows its "TEXT REGION" to be shared. A program's executable file image can indicate, whether its "TEXT REGION" is shareable or not by its magic number.

If allowed, then kernel follows xalloc() algorithm either to find an existing text region of this program (i.e. if some other process already executed this program, its "TEXT REGION" will be already in memory) or to assign a new region to copy "TEXT REGION" of this program in it.
The **xalloc()** algorithm is as follows .....

```
01:  algorithm : xalloc () /* allocate & initialize text region */
02:   input : inode of the executable program's file whose "text region" want to share
03: {
04:   if(executable file does not have separate text & data region)
05:         returns;
06:   if(text region already exists)
07:   {
08:         /* means if inode pointer field in "text region" matches with executable  file's inode
                  (i.e. taken as parameter) */
09:         lock the region;
10:         while (contents of region are yet not ready)
11:         {
12:               increment region's reference count;
13:               unlock region;
14:               sleep(event : until contents of region become ready);
15:               lock region;
16:               decrement region's reference count;
17:         }
18:         attach the region to the process by attachreg;
19:         unlock the region;
20:         return;
21:   }
22:   /* if no such "text region" already exists*/
23:   allocate new text region by allocreg; /* already locked*/
24:   if(inode mode has sticky bit set)
25:         turn "on" the region's sticky flag;
26:   attach the region to the process by using attachreg();
27:   if(file is specifically formatted for paging system)
28:         /* chapter 9 will discuss about this */
      [ Follow the methodology of fork() system call in paging system i.e. satisfy the need dynamically ]
29:   else /* means not formatted for paging system */
30:         read file's text instructions into this text region by loadreg;
31:   change "region protection flag" in per process region table to "read only";
32:   unlock region;
33: }
34:  output : nothing
```

- Kernel first searches "ACTIVE REGION LIST" to (where allocated region are kept) see whether the "TEXT REGION" that process wants, already loaded in memory or not.

  Kernel does this by matching "INODE POINTER" field in each active region with the executable file's inode, that process get as parameter to xalloc().
- If such region exists in "ACTIVE REGION LIST", it first locks the region & checks whether the region is "LOADED IN MEMORY"(note that being present & being loaded are two different

things). If “NOT LOADED INTO THE MEMORY”, process sleeps until region gets ready to load in memory. While doing above things, it accesses the region and hence increments region’s reference count. Then if it find that region is “NOT READY TO LOAD IN MEMORY”, it unlocks the region and then sleep.

When region gets loaded into memory (as it is already present in “ACTIVE LIST”, it is due to some other process hence that process will load it into the memory), it wakes up, accesses the region, and again locks it and decrements region’s reference count, which was incremented before just for checking purpose.

- Then kernel attaches the loaded region to this process by attachreg(), then unlocks the region(the region cannot be kept locked by one process, because other processes may need it for same sharing purpose) and returns.
- If region does not exist in active list then this is the only process, which is going to load it in memory.

  So here kernel have to allocate new region by allocreg(), if the “MODE” field in inode of this file is set to “STICKY BIT” it turns “STICKY BIT” field in this region “ON” and then attaches the region to this process by attachreg(). Here a question may arise where to attach it? This information is available from inode’s data block, which contains “TEXT REGION’S HEADER SECTION” somewhere in 13-member array. This section header there is “VIRTUAL ADDRESS” field, which tell where to attach this region in process’s address space.
- If the executable file is specifically formatted for “PAGING SYSTEM”, then the actions to be taken are explained in the chapter of “MEMORY MANAGEMENT”.
- But if the executable file is not formatted for “PAGING SYSTEM”, then kernel starts reading of file’s “TEXT REGION” and copies read contents into newly allocated “TEXT REGION” by using loadreg(). So now region gets loaded into memory.
- Most importantly, it changes per process, region’s field of this region to “READ ONLY”, which later can cause “PROTECTION FAULT” if it or any other process tries to overwrite on it.
- Finally it unlocks the region (which was locked in allocreg()) and returns.

Some important discussion: - Recall from the allocreg() algorithm, when kernel allocates the region for a file, and if it is not the only process which is accessing the file, (i.e. file’s inode pointer is not null), then kernel increments reference count field of inode of the file whose text region is to be accessed.

This step of allocreg() occurs after the call to exec’s namei() line. Where inode reference count incremented already. But also **Note that** at the end of exec the inode reference count is again decremented. So due to call to allocreg(), the inode reference count will be at least 1, though decremented by exec. If this is not done in allocreg() and if any other process tries to unlink the file, the inode reference count will decrement to 0. but we should not worry about this, because allocreg(), attachreg() & loadreg() had already pulled the required “TEXT REGION” in memory . so even if executable file is not there, our needed “TEXT REGION” is already loaded in the memory.

But a new problem arises here, if a process unlinks the only one occurrence of this file, the global inode table will not have our executable’s file inode in it.

Our process wants this incore inode table entry of file’s inode to assign “INODE POINTER” field in loaded “TEXT REGION”.

So as explained above, due to unlink, if inode’s reference count drops to 0, inode will be released and will appear on free list of inodes and kernel may reassign it to some other file, and thus over text region’s “INODE POINTER” field will point to wrong file’s inode.

Another thing is that, if our process calls exec() again for a new, different program, immediately after prior exec(), then kernel may pickup old file’s inode from the free list (as it is now on free list due to unlink) and thus old file’s inode will appear in new file’s text region.

That is why kernel avoids all above problems by incrementing executable inode’s reference count in allocreg() by checking whether inode pointer is not null. This increment, avoid reassignment of the same in-core inode.

Now when this reference count will decrement (which is incremented by allocreg()) ? : - When process detaches this region, either in exit() or in exec(), the reference count will be decremented. Surprisingly if freereg() is called somewhere in between, it will decrement the reference count of inode for one extra time, but only if the inode does not have its sticky bit mode set.

Above things will get cleared by one example: - Consider the same program of "DATE"....

When exec() is executed kernel allocates a region table entry for its text region, and after exec() completes its inode reference count field of executable file's inode will be left 1. (due to allocreg()) when process exits, this 1 will be decremented to 0 by freereg() in exit(). So no problem!

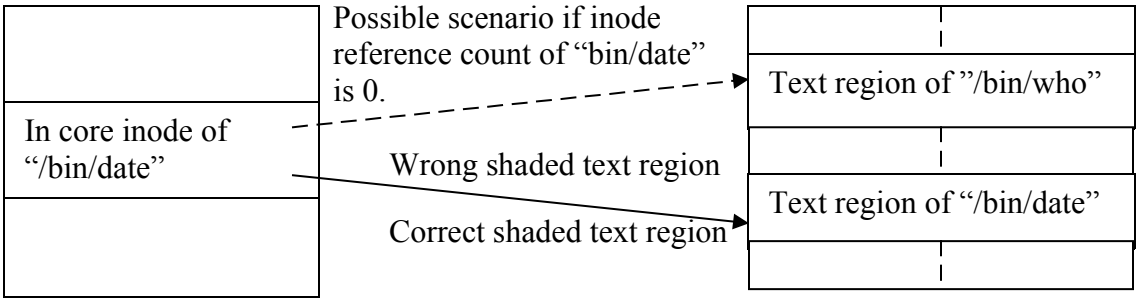
Now consider that there is no "INCREASE RFERENCE COUNT OF INODE OF EXECUTABLE FILE" line in allocreg() and this is the first process which is getting "TEXT REGION".

So when exec() call made, the inode reference count will be 1 after namei() and will become 0 after iput at the end of exec. The program is running. But as its inode reference count is 0, its inode is now on "FREE LIST". Now suppose other process exec the "/BIN/WHO" program and kernel allocates the inode of "/BIN/DATE" to it (as it is now free list of inodes).

Now for sharing purpose when kernel searches the region table it finds entry for "/BIN/DATE" and then mistakenly considers that text of "/BIN/DATE" is the text of "/BIN/WHO", and thus wrong program gets executed.

But this will not happen if allocreg) increments the reference count of inode avoiding reallocation of inode.

The figure for above scenario is as follows…



Global incore inode table Global region table

<u>STICKY BIT</u>: - "SHARING OF TEXT REGION" can decreases the startup time of kernel to start an executable program by exec(). This is done by the "STICKY BIT" feature set.

System administrator can set the file mode to "STICKY BIT" by using chmod() system call to those executable programs which are frequently used.

Thus when a process executes that executable file whose sticky bit is set, the kernel does not release the memory allocated for "TEXT REGION" of that executable file though it detaches it during exit() or exec() even if region reference count drops to 0.

The kernel leaves the "TEXT REGION" intact with inode reference count of that executable file as 1 even though this region is no longer attached to any process.

So in future when another process executes this executable file by exec(), it finds executable file's text region in region table making the startup time for kernel to execute, smaller. Because as "TEXT REGION" is already in memory (i.e. in region table) it does not require to do any I/O.

Even though kernel has swapped this "TEXT REGION" to secondary storage device, then too the kernel requires less time to pull it in memory that to get it in memory by I/O.

Now we will see the conditions when it becomes necessary to remove entries for "STICKY BIT TEXT REGION" from region table. Such conditions are 5 in number...

1. If process opens the file (whose "TEXT REGION" we are taking out) for writing, then text is going to be changed and hence sticky bit is unset and region is removed from region table.

2. If a process changes permission mode of this executable file (whose text region is sticky) by chmod() such that "STICKY BIT" can not remain set. In such cases also the region entry is removed from the region table.
3. If a process unlinks the file, then any process can not call exec() for this executable file because now file is not part of file system. As here onwards "TEXT REGION" is not going to be needed, kernel removes this region from the region from the region table.
4. If a process unmounts the file system on which the executable file was locked, then too, as file's existence is no longer remain in file system, the kernel removes text region entry of it from the region table.
5. If kernel runs "OUT OF SPACE" on secondary storage device (i.e. swap device), then kernel tries make more room by freeing "STICK BIT TEXT REGION ENTRIES" that are currently unused .

   Out of above 5 conditions, removal of "STICKY BIT TEXT REGION" is must in first 2 cases.

   Although it must be noted that in any condition kernel will remove those "STICKY BIT TEXT REGIONS" whose region reference count is 0. Means no any process currently using this region. Otherwise calls to open() (in case 1), call to unlink() (case 3) & call to unmount (case 4) will fail.

What Will Happen IF A Process Executes Itself By Using exec() ? : - This scenario is more complicated. Consider following command...

 # sh script

Here script is a text file having sequence of some commands to be executed sequentially "SH" is the shell's executing program. Thus "SCRIPT" is command line argument to the shell's program sh.

Now recall that when command prompt (i.e. #) appears on the screen, it indicates that shell is running. Thus when we call "SH", it means that shell is executing itself by using exec() system call.

In such cases problem mainly arises during sharing of text region. Here kernel must avoid deadlocks over the inode locks & region locks. Means kernel cannot lock the "OLD" text region, hold the lock and then try to lock the "NEW" region, because here both "OLD" & "NEW" text regions are same region.

So instead of this deadlock, kernel simply leaves the "OLD" text region attached to the process and proceeds. This works because when time comes it will be reused again.

Conclusion on exec() :- It is a bad idea to call exec() only in the calling process. Because as the executable file(called via exec()) is going to overlay over process's address space, the process cannot do anything after completion of exec() because its old address space is lost.

Instead, use fork() to create a child & then call exec() in that child's code. This will cause overlay of child's address space and not that of the parent. So after completion of exec() in child, child will exit and then parent can proceed as usual to do some useful tasks.

Another issue is that if it is better to use "FORK() & EXEC()" one after the other, is it not possible to create such a system call which will internally combine fork() & exec()? It is quite possible but keeping them separate is better to get advantages of fork() without exec(). This separation allows .....

a) Process (i.e. parent &child) can execute independently and can manipulate their stdin, stdout & other file descriptors independently.
b) Such separation also allows use of pipes more skillfully.

The UNIX designers Ritchie & Thompson also support "SEPARATION" rather than combination of these 2 system calls.

The User ID of a Process: - As we know process has process ID. But kernel also gives other 2 Ids to every process. **These 2 ids are...**

(a) Real user ID and

(b) Effective user ID.

- The real user ID identifies the user who is responsible for running the process.
- The effective user ID is used to assign ownership of newly created files. This id is also used to check file access permissions and also to check permission to send signals to other processes via kill().

Out of these 2, kernel allows to change the effective user ID of a process when it executes setuid program by exec() or when setuid() system call is used directly.

Setuid is an executable program which has its setuid bit set in its "PERMISSION MODE" field.

So when a process executes setuid program, kernel sets the effective user ID field in process's U area and in process table to the owner ID of the setuid program file.

For simplicity, though "EFFECTIVE USER ID FIELD" is at two locations i.e. in process's U area & in process table, we will call the "EFFECTIVE USER ID FIELD" in process table as "SAVED USER ID".

The syntax of setuid() system call is...

setuid(uid);

where "UID" is the new user Id to be set on old user ID.

The result of this system call will depend on the current value of "EFFECTIVE USER ID". Means...

a) If currently, the "EFFECTIVE USER ID" of the calling process is super user, then kernel resets both "real user ID" & "effective user ID", both in process table & in process's U area, to uid.
b) If currently, the "EFFECTIVE USER ID" of the calling process is not super user, then kernel resets "EFFECTIVE USER ID" in process's U area to uid if and only if uid is the "REAL USER ID" or if uid is the saved user ID.
c) In all other conditions setuid() fails, and returns on error. **Note that,** generally a process inherits its "REAL USER ID" & "EFFECTIVE USER ID" from its parent when parent calls fork() to create this process. These 2 Ids are maintained across exec system calls.

Following program demonstrates the use of setuid() system call .....

```
#include <fcntl.h>
void main(void)
{
        /* variable declaration */
        int uid, euid, fd_flower, fd_fruit;
        /* code*/
        uid = getuid(); /* get "real user ID" */
        euid = geteuid(); /* get "effective user ID" */
        printf ("uid = %d AND euid = %d\n",uid,euid);
        fd_flower = open("flower.txt", O_RDONLY);
        fd_fruit = open(" fruit.txt", O_RDONLY);
        printf("fd_flower = %d AND fd_fruit = %d\n", fd_flower, fd_fruit);
         /* now call setuid() system call for uid */
         setuid(uid);
        printf("after calling first setuid(%d) : uid = %d AND euid = %d\n", uid, getuid(),getuid());
        fd_flower = open("flower.txt", O_RDONLY);
        fd_fruit = open(" fruit.txt", O_RDONLY);
        printf("fd_flower = %d AND fd_fruit = %d\n", fd_flower, fd_fruit);
        /*again call setuid() system call, but now for euid*/
        setuid(euid);
         printf("after calling second setuid(%d) : uid = %d AND euid =  %d\n",euid,    getuid(),getuid());
```

```
}
```

Assumptions: -

1. After creating executable file of above program, its owner is mango, having uid say 8319.
2. The "SETUID BIT" of above file is "ON".
3. All users have permission to execute above file.
4. Suppose there is an user called Rose whose uid is 5088 and it owns the file flower.txt
5. There is another user called Mango whose uid is (as seen in assumption 1) 8319 and it owns the file fruit.txt
6. Both above files i.e. flower.txt & fruit.txt have "READ ONLY" permission for their owner only.

Now suppose the user Rose executes above program, then the output will be…
Uid = 5088 AND euid = 8319
Fd_flower = -1 AND fd_fruit = 3
After calling first setuid(5088) : uid = 5088 AND euid = 5088
Fd_flower = 4 AND fd_fruit = -1
After calling second setuid(8319) : uid = 5088 AND euid = 8319.

The first line of output: - The getuid() & getuid() system calls return "REAL USER ID" i.e. uid & "EFFECTIVE USER ID" i.e. euid as 5088 & 8319.

As user "ROSE" is now executing the program and it has its uid 5088, the uid = 5088 gets printed.

But ownership of this program's file goes to user "MANGO" which has its uid 8319. As it is owner of this program file, the "EFFECTIVE USER ID" i.e. euid = 8319 is printed.

The second line of output: - Though "ROSE user is the owner of "FLOWER.TXT" file, it can not open its own file and thus fd_flower file descriptor is –1. this is because the program's owner "MANGO" i.e. effective user ID 8319, has no read permission for flower.txt file.

But though "ROSE" executes program, it can open fruit.txt even it is not owner is the fruit.txt file. This is because program's owner is the owner of fruit.txt, which is "MANGO". Thus effective user ID of program works here and "ROSE" can open a file, which is not owned by it. The file descriptor is returned as 3.

The third line of output: - The user "ROSE" is actually executing this program. So "REAL USER ID" is uid of "ROSE" i.e. 5088.

So when setuid() is called with parameter 5088, the "EFFECTIVE USER ID" charges from 8319 to 5088. Thus now both uid & euid are same printing output of 5088 for both uid & euid.

The fourth line of output: - As now "EFFECTIVE USER ID" of this program changed to that of "ROSE" i.e. 5088, now the file flower.txt can be opened by "ROSE" returning file descriptor 4 for this file. But "EFFECTIVE USER ID" 5088 does not have "READ PERMISSION" for fruit.txt and thus it can not be opened returning file descriptor –1.

The fifth line of output: - When setuid() is called again but this time for "EUID" variable which is 8319, it will change "EFFECTIVE USER ID" again to 8319. but user id will not change and will remain 5088.

Note that, when setuid() is called with euid i.e. "EFFECTIVE USER ID", it re-establishes original "EFFECTIVE USER ID" saved in process table as "SAVED USER ID".

Thus final line of output shows that process can use setuid() program/system call to toggle its "EFFECTIVE USER ID" between "REAL USER ID" (i.e. of executer) and "SAVED USER ID" i.e. parameter to setuid().

Now if user "MANGO" executes above program, then the output will be .....

Uid = 8319 AND euid = 8319
Fd_flower = -1 AND fd_fruit = 3
After calling first setuid(8319) : uid = 8319 AND euid = 8319
Fd_flower = -1 AND fd_fruit = 4
After calling second setuid(8319) : uid = 8319 AND euid = 8319.

- Here as "MANGO" itself owner & executer of the program both "REAL USER ID" & "EFFECTIVE USER ID" will always remain the same i.e. 8319.
- The process can never open flower.txt file because "EFFECTIVE USER ID" has no read permission for it(as mango is not owner of flower.txt). But it can open fruit.txt file always because program owner & file owner are same and thus "EFFECTIVE USER ID" has "READ PERMISSION" for the fruit.txt file. **Note that,** the "EFFECTIVE USER ID" stored in process's U area is the result of most recently called setuid program or system call. This is solely responsible for file access permission checking. While the "EFFECTIVE USER ID" saved in process table (i.e. saved user id) is responsible for toggling & resetting "EFFECTIVE USER ID" & restoring it to the original.

  So up till now, we know that "SETUID" can be done by 2 ways, either calling it as a program with exec() or calling it as a system call setuid().

  - The "LOGIN" program is a typical example of using setuid as system call setuid(). By default setuid for login is "ROOT" i.e. the super user and thus "EFFECTIVE USER ID" is root. So when "NAME" & "PASSWORD" is given at login prompt, it invokes setuid() system call to set "REAL USER ID" & "EFFECTIVE USER ID" to that of the user who is trying to login. Finally, means after successful completion of login, shell gets executed whose "REAL USER ID" & "EFFECTIVE USER ID" are set to the login user.
  - The "***mkdir***" command is typical example of using setuid as program. Recall from the topic of mknod() algorithm to create special files such as directory. These we stated that if the user is not super user then return error. So mkdir command, (which internally uses mknod() with some "DIRECTORY CONCERNED IMPROVEMENTS") allows call to it by a process which has its "EFFECTIVE USER ID" as that of super user. Thus it is said that mkdir command is "SETUID USING PROGRAM" owned by root(i.e. super user)

    so first mkdir() (though called by user) runs with "SUPER USER PERMISSIONS", calls mknod() to create the directory and then changes newly created directory's owner & access permissions to the real user. Here "SUPERUSER PERMISSION" to mkdir is given by setuid program which has currently "SUPERUSER" effective user ID. (i.e. root).

So we can say that setuid program is used when doing something with a program which internally calls such a system call, which needs "SUPERUSER'S EFFECTIVE USER ID". E.g. :- ordinary user cannot create directory but when mkdir is called it internally calls mknod.

As seen in growref(), the system call for changing size of process is brk(). Thus brk() system call is mainly used to increase or dense the size of process's data region. The syntax is .....

brk(endds);

Where endds is the value of highest virtual address of the data region of the process. This is also called as "BREAK VALUE".

Alternatively (means instead of brk()), the process also can use sbrk() function which a C lib function which internally calls brk() in it.

Oldendds = sbrk(change);

Where oldendds is break value before call & increment is specified no of bytes by which the region is changed (+ve or –ve)

```
01:  algorithm: brk()
02:  input: new break value
03: {
04:   lock process's data region;
05:   if(region's size is going to increase)
06:   {
07:         if (new region size is beyond system's limit)
08:         {
09:               unlock data region;
10:               return (error);
11:         }
12:   }
13:   change region's size by growreg();
14:   "zero out" addresses in newly added region space;
15:   unlock data region;
16: }
17: output: old break value.
```

Note that ➔ The newly added data space into the grown data region is contiguous with the old data region. Means virtual address space of the process extends continuous into the newly allocated data space.

- Kernel first locks the data region so it can make the changes into it.
- Then if the region's size is going to increase, it checks whether the new size goes beyond the system's limit on the process size. If it is, then it unlocks the region and returns error.
- But if every checks is passed, then it calls growreg() algorithm to increase the size of region. While doing this it allocates new page tables for the data region then goes to process table and increases size of the process in "PROCESS SIZE" field.
  - On "SWAPPING SYSTEM" it also clears the new memory (removes garbage) and initialize all address to 0.
  - If there is no room in memory (i.e. no page tables are available), it swaps the process to get new space. (We will see about it in more details in chapter 9)
  - If process is calling brk() to free the previously allocated space(i.e. decreasing size of region by specifying –ve value in parameter), kernel releases the memory. Now by mistake process accesses "ALREADY FREED SPACE" kernel gives memory fault.
- Finally it unlocks the region and algorithm finishes by returning old break value.

Example → this example shows use of brk()…..

```
# include <signal.h>

/* global function declarations */
void signal_catcher(int);
```

```
/* global variable declarations */
int call_number;
char *cp;
void main(void)
{
        signal(SIGSEGV, signal_catcher);
        cp = sbrk(0);
        printf("original break value was %u\n",cp);
        for(; ;)
                *cp ++ = 1;
}

void signal_catcher(int signal_number)
{
        /* code */
        call_number ++;
        printf("caught signal %d %dth call at address %u\n", signal_number, call_number, cp);
        sbrk(256);
        signal (SIGSEGV, signal_catcher);
}
```

- The signal system call is set to catch "SEGMENTATION VIOLATION" signal (i.e. SEGV).
- Calling sbrk() first time with 0 is just to get the previous "BREAK VALUE". So no size change takes place here.
- Then it infinitely loops incrementing the character pointer & its contents.
  The *cp ++ = 1 can be splitted in 2 statements as...
  ***cp = 1;**
  cp ++;
  The loop continues until process tries to write on such address, which is beyond its data region. When it goes beyond the data region, signal "SEGMENTATION VIOLATION" will occur and signal_catcher function will get executed.
- In signal catcher function it increments a global variable (initialized to 0 automatically) to get the loop counter.
- Then it prints signal number, loop counter and current address.
- It again calls sbrk() function, but this time not to catch just break value, but to actually increase the size of region by 256 bytes.
- And then prepare to catch the same signal again. During execution, original break value will get printed and loop starts.When printf() in signal_catcher() gets printed, it is an indication that current data region is exhausted and process is trying to write beyond it so signal of "SEGMENT VIOLATION" appears, sbrk() increases size by 256 bytes and process again enters in infinite loop of main.
- The whole above procedure will repeat again & again.

About changing stack region size: - As explained previously in growreg, growth of stack takes place automatically when stock overflows. The internal algorithm for growth of stack is quite similar to brk(). Overflowing of stack usually takes place as stack pushes more & more frames onto it during execution of process.

When stack overflows beyond its current limit, it causes memory fault because new frame tries to get added outside the process's address space. Kernel determines this fault and its cause by comparing the current value of stack pointer (faulted) with that of the original size of the stack region.

Then kernel allocates new space for the stack region exactly as shown in brk(). Now process can continue with newly grown stack.

THE SHELL: - As we know when we login to UNIX system, ultimately shell is started. As we work in shell(up till we change the default shell) continuously executing different programs, we can imagine important things ……..

- Shell is such a program, which can behave like parent of all programs we run from shell. Obviously programs those we execute from shell prompt, became children of shell.
- As shell continuously run & reads from command line i.e. whatever we type, it must have a loop structure.
- When we do not use "REDIRECTION" or "PIPE", every input is taken from stdin, every output goes to stdout & every error goes to stderr, which are "STANDARD TERMINAL".
- When we use "REDIRECTION" to redirect the output to some file, that file is created by the shell with our usual creat() system call and then output is written by write().
- When we use "PIPE" to synchronize two commands/program the shell becomes grandparent, the 2nd command (i.e. right sided to pipe) becomes parent and the 1st command (i.e. left side of the pipe) becomes the child (i.e. child to parent but grandchild to shell) and so on.
- By default shell's processing is synchronous, but by using "**&**" (ampersand operator) we can push a program to perform background task and then start a new program while previous program is already running in background.
- Some commands are "INTERNAL" to shell, means those commands do not have separate executable files. Hence to execute such commands exec() is not needed. But some commands are "EXTERNAL" to shell. Means they are not "BUILT IN" in shell. Such commands have their executable files usually located in "BIN" directories of "ROOT" &"USR". Obviously such commands require exec system call to execute them.
- Reading of "USER GIVEN INPUT" by the shell from the shell prompt is done by usual read() system call whose first parameter is file descriptor- stdin(unless some other is mentioned by redirection). Some thing about output which uses write() with first parameter as about which uses write() with first parameter as file descriptor – stdout or in case of error – stderr. With this brief background knowledge, now we can see the algorithm for "MAIN LOOP" of shell…
  No new process is created for internal command like cd.

```
/*  read command line until "eof" occurs  */
while(read (stdin, buffer, numchars))
{
        /* parse the command line */
        if(/* command line contains "&" operator */)
                amper = 1;
        else
                amper = 0;
        /* if the command is not "internal command" i.e. if command is an executable program*/
        if(fork() = = 0)
        {
                /* if redirection is used */
                /* here only "redirected output" is considered */
                if(/* redirect the output */)
                {
                        fd = creat(newstyle, fmask);
                        close(stdout);
                        dup(fd);
                        close(fd);
```

```
                    /* stdout is now redirected to newly created output file */
            }
            if (/* pipe is used */)
                    {
                            pipe(filedes);
                            if(fork () = = 0)
                                    {
                                            /* first component of command line */
                                            close(stdout);
                                            dup(filedes[1]);
                                            close(filedes[1]);
                                            close(filedes[0]);
                                            /* stdout will now go to writer pipe */
                                            execlp(command 1, command 1, 0);
                                    }
                            /* Second component of command line */
                            close(stdin);
                            dup(filedes[0]);
                            close(filedes[0]);
                            close(filedes[1]);
                    /* standard input will now come from "reader" pipe */
                    }
                    execve(command2, command2, 0);
    }
        /* shell (i.e. grand father continues from here */
        if (amper = = 0) /* means synchronous, then wait for child's exit */
        returned = wait(& status);
}
```

- Shell reads a command line from "STANDARD INPUT" (stdin) and using "SOME FIXED RULES" interprets & rearranges it for passing.
- If shell interprets the input string from command line as its "INTERNAL COMMAND", it executes it internally, without creating a child process (thus not shown in algorithm).
- If it does not find the input string as one of its "INTERNAL COMMAND", it interprets the command is an executable program.
- For such executable program shell forks a child process and executes it by using execve() system call. In the algorithm this call to execve() appears quite for but reason will be clear later.
- Before calling execve(), it ensures that whether "REDIRECTION" is used. Though it handles both "REDIRECTED INPUT" & "REDIRECTED OUTPUT", algorithm shows only how to use "REDIRECTED OUTPUT"....

"REDIRECTED INPUT" is done for stdin in similar ways.

Suppose output is redirected to a file then .....

- It creates the file (as per user's decided name on command line) by using creat() system call. If something goes wrong here (i.e. wrong permissions etc) the child will exit immediately.
- As output is going inside the file, it closes stdout and dups file descriptor of file so that it will take place of stdout and output will thus go to file. As file descriptor is now not needed (because it is already duped), it closes file descriptor of created file too.
- Before calling execve() it ensures whether "PIPE" is used. On command line pipe has a form like...

Command 1 | command 2

Where command 1 is always an "OUTPUT GIVING" command and command 2 is always "INPUT TAKING" command. As shell's loop is a "RENDER LOOP", its child holds command2, because it is "INPUT TAKING" i.e. reader sided command.

But for command 1, it again forks a child process (which child to the shell's child), which now works for command 1, as .....

- As command 1 is output giving command to pipe, it closes "STDOUT", then dups "WRITER PIPE" and then closes unnecessary pipe descriptors.
- Now whatever be the output of command 1, it will go to "WRITER PIPE".
- Then it uses execlp() function to execute command1, whose output will go to "WRITER PIPE".

The two programs with pipe i.e. command 1 & command 2 run asynchronously to each other

For command2 as it is a "READER PROCESS" (or "INPUT TAKING" process, it works as...)

- It closes "STDIN", dupes "READER PIPE" and then closes un-necessary pipe descriptors.
- Now whatever command1 will read, it will read from "READER PIPE".
- Then it uses execve() system call to execute command1, which will read from "READER PIPE".

Note → As we said before, exec() of shell's child appears for away. This execve() works for that purpose. Means this call to execve() is common to both situations.....

(1) When executing a command without any pipe or redirection.

(2) When executing a command with pipe or redirection or both.

That is why execve() flavor is used. Reason will be clear later.

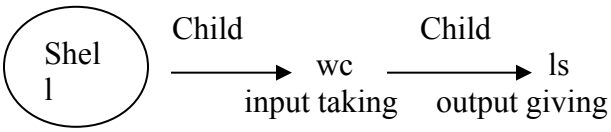
- After completing the code for shell's child, now following code is for shell. Where it checks whether "AMPERSAND" is used or not. If it is used, child will continue in background and shell will read next command line by coming back to "WHOLE", indicating asynchronous processing.

  But if "AMPERSAND" is not used, then shell will wait until child's exit and after child's exit will loop back to "WHILE" indicating synchronous processing.

Example

 # ls | wc

Here though "ls" appears first & wc appears next to it, the rearrangement by shell is done like...



Known flavors *:- execl, execle, execlp, execlpe, execv, execve, execvp, execvpe.*

In shell's main loop execlp & execve are used. Reason for that can be understood by meaning of suffixes.

About flavors of exec() :- As described in exec, library has several flavors for it. All start with common letters exec but different suffixes are l, e, v, p, which are either used individually or in combination. Different flavors of exec() have slightly different functioning but ultimately they all execute a program given to them as their parameter.

Meanings of these suffixes are .....

- l → This suffix is used to indicate that command line arguments are passed individually to the function and hence it is used in that situation where number of command line arguments (i.e. number of parameters) is fixed.
- v → This suffix is used indicating "VARIABLE", means either command line arguments (i.e. parameters) or environment array is of variable length(i.e. not fixed).

- e → This suffix is used when envp parameters i.e. environment array is specifically given by the user to the exec.
- p → This suffix is used when "PATH" environment variable is going to be used for finding the executable file in exec.

System Boot And Init Process

The operation to initialize a system from an inactive state is known as Booting. Usually Administrator "BOOTS" the system. The procedure of "BOOTING" vary machine to machine but all machine follow a common goal and that is " TO GET A COPY OF OPERATING SYSTEM INTO MACHINE"S MEMORY AND START EXECUTING IT HERE".

This is done in a series of stages of using functions, which work independently, and which internally calls another function, which calls another and so on. Thus this procedure is given a name bootstrap.

So when computer's switch is mode "ON"

↓

Loads a "BOOT STRAP PROGRAM" from ROM boot strap program it turn has few instructions that instruct the machine to execute another program

↓

In Unix, boot strap ultimately reads the "BOOT BLOCK" (i.e. block 0) of a disk and loads contents of that block into memory (usually boot loader).

↓

The program is this boot block (which is now in memory) loads kernel (usually from the file /Unix) from the file system (on disk) into the memory.

↓

After kernel gets loaded into the memory, the boot program transfers control to the "START ADDRESS OF KERNEL" (recall that "PRIMARY HEADER" of executable file has "START ADDRESS" i.e. from where to start the executable program).

↓

Now kernel starts running with the algorithm "START".

↓

In the algorithm of "START", kernel initializes its data structures e.g.: - Constructing linked lists of free buffers and free inodes, constructing hash queues of buffers and inodes (allocated), initializing region structures, page table entries etc.

↓

After completing above "INITIALIZATION & CONSTRUCTION" phase, kernel mounts the root file system onto the root i.e. on "/".

↓

Then it creates hard coded environment for execution of process 0 such as creating U area for it, initializing 0^{th} slot in process table for process 0 and making "ROOT" as the current directory & current root of the process 0.

↓

After completing “SETTING UP ENVIRNMENT” for execution of process 0, the system is said to be running as the process 0.

↓

Now process 0 calls fork(), invoking fork() algorithm directly from the kernel, because process 0 is right now in kernel mode (and remember it always runs in kernel mode only)

↓

This fork of process 0, creates a child process known as process 1 or init. Process 1 is right now in kernel mode. It creates its “USER LEVEL CONTEXT” by allocating a regions and then attaching these regions to its own address space. Then it grows these regions to its proper size and copies code from kernel address space into these new regions. This code now from the “USER LEVEL CONTEXT” of process 1.

↓

Process 1 then sets up its “SAVED USER REGISTER CONTEXT” and then returns from kernel mode to user mode, and starts executing the code, which it just copied from kernel. So now process 1 becomes “THE USER LEVEL PROCESS”. The “TEXT REGION” or the code of process 1(which is just copied from kernel) contains a call to exec() system call to execute the program “/ETC/INIT”. So process 1 executes this program. That is why process 1 is called as “INIT” because it is responsible for initialization of new processes.

↓

So now control gets transferred to “INIT PROCESS” (or say init program executed by process 1) this process is also called as “PROCESS DISPATCHER”. Algorithm of init first opens & reads the file “/ETC/INITTAB” to see which processes the init should spawn (the term “SPAWN” really means “LAYING EGGS” so here it means “CREATING MORE & MORE CHILDREN”). The entry in the inittab file contains id, state, action and process specification & comment if any E.g.: - A comment stocks by operator. ***(inittab means “INIT TABLE”)***

↓

id	state	action	process specification	comment
46	2	respawn	/etc/getty consol consol	# commit

↓

So init reads one entry at a time and if it finds that the “STATE” (single user, multiuser etc.) for which “INIT” was invoked matches with the “STATE” in the line of this file, then it invokes that process (by fork() and then execl()) which is written in “PROCESS SPECIFIATION” field of line in that file.

State 1 → single user
State 2 → multiuser

↓

From this file, init usually spawns “GETTY”: program which monitors “TERMINALS” in UNIX and which does “LOGIN” from the user and finally executes shell.

↓

Now shell takes command. But mean while, init calls wait() which monitors death of either its child or death of "ORPHAN" processes(as seen in exit)

What happens to process 0, after forking process 1 ?

After creating process1, now process 0 & process 1 are completely independent and process 1 proceeds further system startup as explained above.

Meanwhile process 0 (as it works only in kernel made) executes some "KERNEL ONLY PROCESS" like vhand (which is a page re-claiming) process. And finally the process 0 executes algorithm of "SWAPPER" and then become & called as "SWAPPER PROCESS".

Algorithm of "START": -

```
01:  Algorithm: start /* system startup procedure */
02:  Input: nothing
03:  {
04:          initialize all kernel data structures;
05:          mount of root; /* called "pseudo" because real is done later */
06:          create "hard coded" environment for process 0;
07:          fork process 1 by kernel's internal fork():
08:          {
09:                  /*process 1 executing code */
10:                  allocate region;
11:                  attach region to its address space;
12:                  grow region to accommodate possible incoming code;
13:                  copy data & code from kernel's address space to above newly grown region;
14:                  change mode i.e. from kernel mode to user mode;
15:                  execute "/etc/init" by exec ();
16:          }
17:          /* code for process 0 only */
18:          fork kernel processes such as vhand ();
19:          execute algorithm "swapper" to become "swapper process";
20:  }
21: output: nothing
```

Why kernel copies code of process 1 from kernel's space to process 1's address space? Why it cannot call directly exec() for that code ?

- Execution by exec() is done in user mode. As process 1 is now in user mode it can call exec() as exec() system call is designed to work for user mode. If kernel calls exec(), then the existing code of exec() will become more complicated for user mode. Or two versions of exec() would have been made, one for kernel and one for user. The makes loss of transparency.

  Another thing is that if exec() is to be called in kernel, then passing of path of /etc/init would have to be done twice once in kernel mode & then in user mode.

Algorithm of "INIT": -

```
01:  algorithm: init /* initialize the process 1 of the system */
02:  input: nothing
03:  {
```

```
04:   fd = open("/etc/inittab", O_RDONLY);
05:   while (read each line (fd, buffer))
06:   {
07:         /* read every line of file */
08:         if(process 1's invoked state != state given in line)
09:               continue; /* go back to white */
10:         /* when "state" matches */
11:         if(fork() = = 0)
12:         {
13:               execlp("process specified in line");
14:               exit();
15:         }
16:         /* loop back to white*/
17:   }/* means it executes all processes in all lines & then come out */
18:   while ((id = wait((int *) 0)) ! = -1)
19:   {/* check whether spawned child is dead, then either respawn it or just continue */
20:   }
21: }
22: output: nothing
```

Types of processes in UNIX system: - UNIX system has 3 types of processes .....

(a) User processes.
(b) Daemon processes.
(c) Kernel processes.

User processes: - Most processes in UNIX are of this type. Which are associated with Uses at terminal.

Daemon process: - These are not associated with any user. They do system-wide functions, such as administration and control of networks, execution of time dependent activities, line printer spooling etc. the process "INIT" can spawn these processes which then exist throughout the life time of system. Occasionally users also can spawn them. They might be like user processes when they run in user mode and make system calls just like user processes do.

Kernel processes: - They execute only in kernel made. Process 0, for example spawns kernel process vhand (as seen before) and process 0 becomes "swapper process". As their work also "SYSTEM-WIDE" they are similar to daemon processes. But better than daemon processes, they have property or ability to access to kernel algorithms and data structures directly because kernel process's code is a part of kernel itself. Thus kernel processes are very powerful but not flexible as daemon processes because daemon process's code is easily changeable without touching the kernel. While if kernel process's code is changed, then whole kernel has to be recompiled & rebuild.