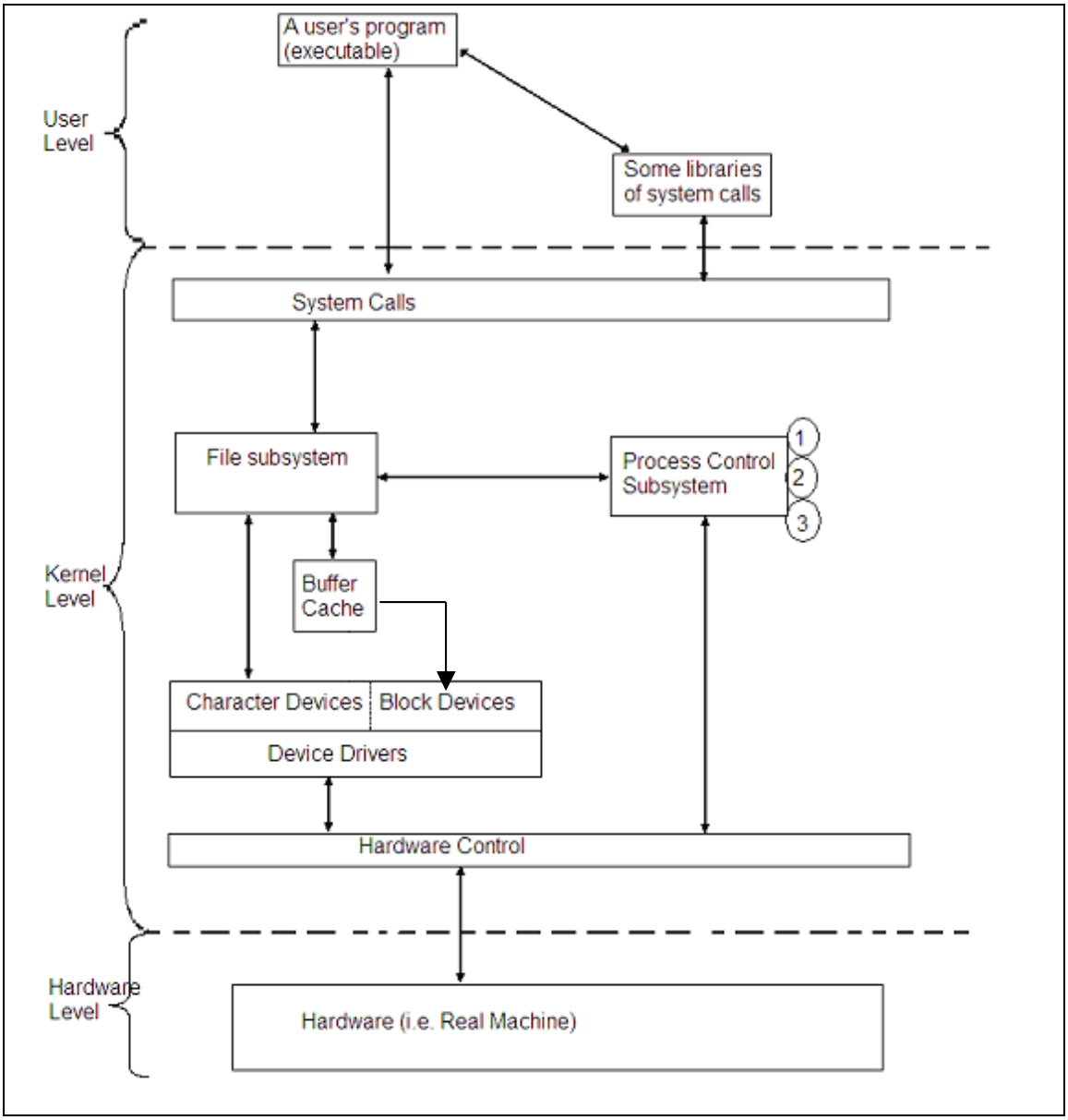
2ND CHAPTER THE KERNEL

UNIX System Architecture



Block Diagram Of *UNIX Kernel*

Tip : *Process Control Subsystem* includes 3 mechanisms –
1) Inter-Process Communication (IPC) 2) Scheduler 3) Memory Management

Though, many times it is said that, Operating System is made up of *Kernel* & *Shell*, but in fact, *Shell* is only command interpreter, which allows communication of you & your system's *Kernel*. The real Operating System is thus ***Kernel***. So it is better to say, "Operating **System is *Kernel* and *Kernel* is Operating System**".

The previous figure gives the block diagram of *UNIX Kernel*.

There are only two central concepts to *UNIX* system:

1) ***File***
2) ***Process***

So, it is said that, "*UNIX* system creates an illusion in which it is show that, "*File has space and Process has life*". Though it is not real, *UNIX* is designed to achieve this goal.

Thus system designers included these two major components i.e. file & process, into *Kernel* as subsystems. Thus *UNIX Kernel* has *File Subsystem* & *Process Control Subsystem*.

Now to analyze the architecture of *UNIX Kernel*, we will assume that there is compiled program, which user wants to execute…

User executes a program.

↓

Now running program becomes an entity, so called Process.

If the program of that process is written in Assembly language, then it can directly communicate with the system calls. Or, means if program of that process is not written in assembly language, but written in some high/middle level language, like C, then mostly, instead of direct communication with system calls, process links with a library (which contains high level functions to get communicated with the system calls, not by row method by a sophisticated method) and then library communicates with the system calls. (These libraries are linked with the program at compile time. Hence they become part of the program's code. This linking referred as static linking). Here we end with the "User Level".

The "system calls" are mainly of two types, **One**, those deal with File Subsystem (like ***open(), close(), read(), write(), chmod, chown()***, etc). The file subsystem is that, which is responsible for allocation on disk space to file, administration of "data filled" & "free" spaces on disk, controlling file access, retrieving file data, etc. Thus when a process communicates with the system calls, and system calls are "file subsystem" specific, then above services are guaranteed.

Two, those deal with *Process Control Subsystem* (like ***fork(), exec(), exit(), wait(), brk(), signal()***, etc). The *Process Control Subsystem* is that, which is responsible for Process synchronization & 3 things shown in figure, i.e. Inter-process communication, process scheduling and memory management.

- ✓ The Inter-Process communication part of this subsystem do inter-process communication between different processes, which may be of asynchronous type signaling or may be of synchronous message transmission.
- ✓ The scheduler part of this subsystem allocates (gives) CPU to different processes. This part allows running of a process until it is closed (means free the CPU) by the user or until its given time slice gets finished. After any of the above condition occurs (means a process is either closed or its time slice finishes), schedules chooses the highest priority process (if available in memory) to run. If the 2nd condition occurs, means a process is kept waiting due to its end of time slice, then this process is restarted when it becomes the highest priority process.
- ✓ The Memory Management part of this subsystem controls the allocation of memory. Means if at any time, the system does not have enough physical memory (RAM) to run all available processes, then this part moves the process between main memory and secondary memory so that all available processes should get the chance to execute. This memory management is done either by "swapping" or by "demand paging" methods.

Finally *Process Control Subsystem* communicates with "*Hardware Control*".

The *File Subsystem* & the *Process Control Subsystem*, can interact with each other (hence arrow is shown in between these two blocks in the figure) when loading an executable file into memory for execution. Here *Process Control Subsystem* communicates with the *File Subsystem* to take the file and then it loads & read the file into the memory before starting its execution.

After completing the consideration of *File Subsystem* & *Process Control Subsystem* blocks in the figure, to back to *File Subsystem Block*.

The *File Subsystem* communicates with the "raw" devices via device drivers which are also called as "character devices" (such as printer). These character devices then use their own device drivers to drive the device.

The *File Subsystem* also communicates with "Block" devices and access files on these devices by a special mechanism called as "***Buffering Mechanism***". This mechanism controls the flow of data from the devices to the system and from the system to the devices. These Block devices are "random access storage device" on their own or device drivers of them show them as "random access storage device", though they are not.

Finally, the Device Drivers (both of "Block" devices or of "Character" devices) communicate with "*Hardware Control*". [***Device Drivers***: are *Kernel* programs which control the operation of peripheral devices (Here devices may be of "Block" or "Character".)]

So now we arrive at "*Hardware Control*" block of *Kernel* diagram, where *Kernel* level ends.

The "*Hardware Control*" part of *Kernel* is responsible for hardware interrupts and for communicating with the real hardware (or say real machine).

At the end, "*Hardware Control*" part of *Kernel* directly communicates with the hardware or real machine to do necessary task. This is the hardware level and it ends here only.

Introduction To System Concepts

1) *File Subsystem*
2) *Process Control Subsystem*: -
    a) Process.
    b) Context of a process.
    c) Process status.
    d) State transitions.
    e) Sleep & Wake-up.
3) *Kernel Data Structures*.

I) File Subsystem :

In *UNIX* everything is "file". Internally a file is represented by ***Inode***. ***Inode*** contains "disk layout of the file" and other file related information like the file owner, file access permissions, file access time, etc. *Inode* really means "***Index Node***". Every file has one *Inode*, but file itself may have many names. Though file can have multiple names, each name is mapped into the same *Inode* using link.

When a process (i.e. running program) accesses a file for *read/write/execute*, by its name, e.g.: Suppose a file has its destination ***/vijay/temp/temp.txt***, then *Kernel* travels the path ***/vijay***, then ***/vijay/temp*** and finally ***/vijay/temp/temp.txt***. Means traveling takes place one "path component" at a time. When it reaches the final destination, it checks file access permissions (i.e. *read*, *write*, *execute*) and if access permission permits to the path of the file, it ultimately retrieves the *Inode* at the required file. E.g. ***open ("/vijay/temp/temp.txt", 1) ;***

Similarly, when a process creates a new file, *Kernel* first assigns a new, unused *Inode* for that newly created file.

All *Inodes* are stored in file system. File system is stored on disk. A disk may be partitioned into several file systems. But *Kernel* does not deal with the physical disk. Instead it deals with the logical disk. Conversion of *Physical Disk Layout* into *Logical Disk Layout* is done by disk drivers (provided by vendor). So, in other words, disk drivers convert *physical disk addresses* into *logical disk addresses* and *Kernel* deals with *logical disk/device addresses* only.

Now the *file system* is actually a sequence of such logical device, segmented into logical blocks. Each block is made up of either **512** or **1024** or **2048** bytes (depending on which file system is used) or any other multiple of 512. Note that the block size is same on a single file system. Means if a file system uses 1024 byte division method then all logical blocks on this file system will be of 1024 bytes. (But different files systems on same disk may differ in their block size). When *Kernel* accesses or creates a file, it is read into main memory. So, larger the logical block size, fast will be the process of file accessing, because disk operation are initiated by *Kernel* by sending message to disk driver. As *Kernel* deals with the disk at logical level, if a file system has each block of 512 byte size, it will send message to read a 1 KB file, twice (512 each). But if a file system is made up of 1024 byte sized blocks, then same above operation will be done once only. This does not mean that maintaining for block size is always worth, because large block size decreases storage capacity.

Now we know how file system is created and stored. We will now turn to explore the parts of a file system.

- ★ The non-dotted appearance of boot block & super block indicates that they are of fixed size, right from the time of system installation.
- ★ The dotted appearance of *Inode* block and data block indicates that they are of variable size. Means their size depends on number of files. (And as we can delete or create files as per our will, these blocks are of variable size).

The file system structure contains 4 parts: ...

a) ***Boot Block***: - This is the beginning of file system. On physical or logical level, this is typically the first sector of logical disk. This block usually contains a small program known as *bootstrap program*. When machine boots, this program is executed by ***ROM*** to load the operating system. Every file system has its own boot block. But if your disk has multiple file systems under *UNIX*, only one boot block of one file system contains bootstrap program. Remaining file system's boot blocks are kept empty.

b) ***Super Block: -*** These blocks keep information about: - 1) how large the file system is... 2) how many files this file system can store... 3) where is the free space on this file system, etc.

c) ***Inode List: -*** As stated before, every file has its own, unique *Inode*. Obviously size of this block will be depending upon number of files in this file system. And obviously this block is capable of expanding (when we add new files to our file system) or capable of shrinking (when we delete old files from our file system). System administrator gives a default Inode list size at the time of installation. As seen before, *Inode* is index based (i.e. array), the *Kernel* accesses the *Inode* by index. And as it is a list (similar to link list), it has a header node called as *root Inode*,

which contains the directory structure of the file system. Some contents of other *Inode*s are explained on before.

d) ***Data Block: -*** Up till now we know that, boot block contains only bootstrap program (or empty), super block contains information about file system, *and Inode* list contains information about each file in the file system. Then a question may arise, where is the file data? The data of every file is kept in this data block. As it is concerned with number of files, its size is also variable i.e. expandable or shrinkable. Note that, when *Kernel* allocates a part of data block to a file, then that part of data block belongs one and only one file.

Now we will turn back to our discussion of "how *Kernel* accesses a file?" When system boots, the *Inode* list block is read from the disk and pulled out into the ***RAM*** (main memory) from which *Kernel* can now access the *Inode* of a file. The memory representation of the *Inode* list block is called as *Inode Table* (obviously array).

For file accessing mechanism, *Kernel* has other 2 data structures….

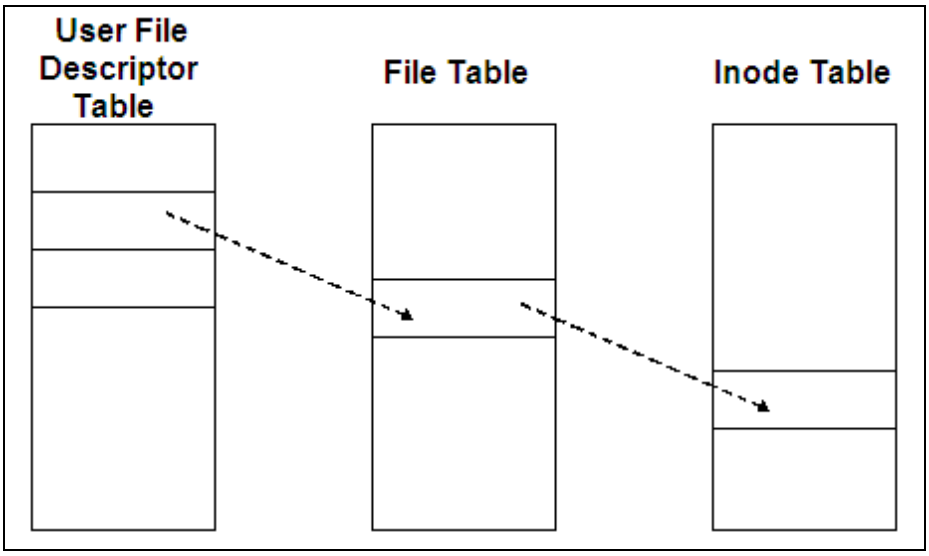
a) ***File Table***: - This table contains byte offset of the file, i.e. from where to begin the next read or write operation. It also contains the file access rights for the process, which is currently opening or reading or writing the file.

b) ***User File Descriptor Table: -*** This contains identifiers of all open files for a single process (as indexes).

Note that, "*File Table*" is a global data structure, while "*User File Descriptor Table*" is allocated to each process (i.e. local).

So, in all file access mechanism needs 3 data structures:

a) A RAM image of *Inode* list block i.e. *Inode Table*.
b) *File Table* and
c) *User File Descriptor Table*.



Whenever a new file is created, entries are made into all these three tables and whenever a file is deleted, entries from all these three tables are removed. At every time of file access (creating, opening, reading, writing and executing), these 3 tables are pulled into RAM, respective changes are made and at the time to shutdown, then changes are saved back to disk.

Thus when *Kernel* creates or opens a file, it returns a *file descriptor*, which is an index (number) from the *File Descriptor Table*. When *Kernel* reads or writes a file, it uses this file descriptor to access *User File Descriptor Table*, from *File Table*, it gets another pointer, by which it can search into *Inode Table* for that specific file. And from *Inode* it finally finds the actual data located in that specific file.

II) Process Control Subsystem :

As stated before, process is the execution state of the program. Many processes may execute simultaneously in *UNIX* like multi-processing system. And also there may be many processes of a single program. A process can read its data and *Stack* section in memory, but can not read or write to another process's data and *Stack* section.

Processes can communicate with each other by using system calls (the method so called is Inter-Process Communication).

A) Process

The *UNIX* system creates every process by using *fork()* system call. The first process, created when system boots is so called ***Process 0***, which then uses *fork()* system call to create ***Process 1***, which then uses *fork()* system call to create ***Process 2*** and so on.

The process which uses *fork()* system call (i.e. to create new process) is called as ***Parent Process*** and the newly created process is called as ***Child Process***.

Obviously every *Child Process* has one and only one *Parent Process*, but a *Parent Process* may have one or many *Child Processes* (depending on how many times the *Parent Process* used *fork()* system calls).

Kernel identifies every process by a *Process ID number* (PID), obviously unique to every process.

Interesting point is that, who will create *Process 0*, (i.e. who will use *fork()* system call to create *Process 0?*). No! Nobody calls *fork()* to create *Process 0*. Instead it is created "manually" without using *fork()* and is created as it is, when the system boots.

Then this *Process 0* calls *fork()* to create *Process 1* and then comes out of the picture by converting itself into just a "***swapper process***".

Now *Process 1* takes charge of all the system and hence called as ***init process***. This then calls *fork()* to create *Process 2* and story goes on. Thus "***init***" process becomes most ancestor process for all other processes and has special relationship with all its child...sub-child...sub-sub-child (and so on) processes. This relationship (i.e. special one) is important and we will consider it afterwards.

We know that process is nothing but "executing state" of a program. In other words, program is nothing but a bunch of code. When it gets executed, it is launched in main memory (RAM) and then we call it as *process*.

So to be a process, the program must be in compiled & linked form which is called as ***Executable Code Of A Program*** (in *DOS* & *Windows,* called as "***.exe***" file and in *UNIX* environment called as "***.o"*** file.)

Now an important question arises, **how Operating System pulls an executable program into RAM and converts it into a process?** Or in other words, **what "executable program" has so special contents, so that Operating System can realize it as valid program to convert it into a process?**

For the answer, we should examine contents of an executable program.

First we will take example of "C program".

A low level File Copy Program naming copy.o

```
#include<stdio.h>
#include<fcnl.h> /* for low level file I/O */
int TempGlobalData = 1 ; /* Global Variable */
char buffer [2048] ; /* buffer */

int main ( int argc, char * argv [ ] )
{
        void copy ( int, int ) ; /* prototype of copy() */
```

```
        /* Local variables */
        int OldFile, NewFile ;

        if ( argc != 3 )
        {
                printf ( "Need at least 2 arguments for copy \n" ) ;
                exit (1) ;
        }
        OldFile = open ( argv [1] , O_RDONLY ) ; /* open for read only */
        if ( OldFile == -1 ) /* error checking & exit */
        {       printf ( "Can not open %s file \n", argv [1] ) ;
                exit (1) ;
        }
        NewFile = creat ( argv[2], 0666 ) ; /* explained later */
        if ( NewFile == -1 ) /* error checking & exit */
        {
                printf ( "Can not crate %s file \n", argv[2] ) ;
                exit (1) ;
        }
        /* file copy */
        copy ( OldFile, NewFile ) ;
        exit (0) ;
}

/* User defined copy() */
void copy ( int old, int new )
{
        /* local variables */
        int count ;
        /* code */
        while ( ( count = read ( old, buffer, sizeof ( buffer ) ) ) > 0 )
        {
                write ( new, buffer, count ) ;
        }
}
```

First we will explain the program:

- ✓ At the most beginning, we included 2 files ***stdio.h*** (for standard input/output functions) and ***fcntl.h*** (for low level file input/output functions) by ***#include <...>*** preprocessor directives.
- ✓ Then we declared 2 global variables. **One** is a character *buffer* i.e. character array of 2048 size. (here 2048 represents 1024 + 1024 bytes i.e. 2 KB). **Second** is an integer variable ***TempGlobalData***, which is initialized to ***1***. This variable is used to explain some concepts and has no use in the program.
- ✓ This is a command line argument program, hence ***main()*** has ***argc*** & ***argv*** parameters. We also can use ***main()*** returning ***void***, but here it deliberately used as returning ***int***, for explaining the ***exit status*** of a program.
- ✓ In the body of ***main()***....
  a) We declared two local ***int*** variables **OldFile** & **NewFile** to hold the file descriptors, returned by ***open()*** & ***creat()*** system calls.
  b) Then we declared command line argument number (i.e. ***argc***). It should be 3, one for program's name, second for the source file name (to be read) and third for the target file

name (on which we will write the read file). Thus if ***argc*** is not equal to 3, we will display the error message by ***printf()*** and we will exit the program with exit status ***1***, with ***exit (1)***. ***Note :*** When exit status is ***0***, then the program is terminated normally and successfully, when exit status is non-zero, then program is terminated with some error. The *UNIX* system calls use ***-1*** by default to indicate that system calls fails. Here "we" are exiting the program (not any system call is exiting), hence we used ***1*** in ***exit()***.

c) Then we opened the source file i.e. ***argv [1]*** (because ***argv [0]*** is program's name, ***argv [1]*** is source file's name and ***argv [2]*** is target file's name) by using ***open()*** system call. As we don't want to write to this source file and as it is already present on disk, we used ***O_RDONLY*** flag indicating that we want to use this file for reading purpose only. As *open()* is a system call, on success it will return an integer which is called as file descriptor (this is the number that *Kernel* uses to search for a file in *User File Descriptor Table*). This file descriptor number is unique for a particular file and has value other than -1. On failure, *open()* system call returns ***-1*** as file descriptor which can use for error checking. Thus we checked next if file descriptor ***OldFile*** is valid or not. And if invalid (i.e. ***-1***) we output a message and exit with ***1***.

d) If *open()* succeeds, we proceed further to create the target file (on which the read file is to be written) by *creat()* system call. (***Note that***, this system call is ***creat()*** and **not create()**). We created the target file ***argv [2]***, with ***0666*** permission parameter. Recall from (Chapter 1), here ***6*** represents ***read*** i.e. ***4*** plus ***write*** i.e. ***2*** *octal notations*. Allowing all (*first 6 for owner*, *next 6 for group-owners* & last 6 for all others). The initial 0 is for octal number. Again if *creat()* succeeds it returns unique file descriptor in ***NewFile*** variable and on failure it returns ***-1***. We checked this variable and if ***-1***, we output a message and exit with ***1***.

e) If *creat()* succeeds, we will call *copy()* function. This function is defined by us (hence it is a ***UDF***). It takes the two valid file descriptors, first that is for source file (i.e. ***OldFile***) and second for target file (i.e. ***NewFile***).

f) At last we exit with ***0***, which indicates O.S. that our program is completed and the exiting status is normal and successful hence 0.

✓ After *main()* we write body of our ***UDF*** *copy()* in which we declared a local integer variable ***count*** and then used "***while***" loop, in which we called *read()* system call, which will return number of read bytes (of source file) into ***count*** variable and will put actual read data into globally declared buffer. On success or otherwise will return ***-1*** or ***0*** when *read()* fails or when source file reading completes respectively. Hence the "***while loop***" is valid until the ***count*** variable is greater than ***0***.

Thus until the ***count*** variable is greater than ***0*** (means it is returning bytes read), the *write()* function call (which is used inside the loop) will write data from buffer to the new file. The number of written bytes are equal to ***count*** variable.

✓ When we call this program (i.e. when we will execute this program) from command line, the command line will be something like....

 # copy source.txt target.txt

Note the number of strings in command line. It is 3 that we have checked in ***argc***. Obviously in *main()*, ***argv [0]*** will be string "***copy***". ***argv [1]*** will be "***source.txt***" and ***argv [2]*** will be "***target.txt***".

Also ***OldFile*** variable will be file descriptor of the source.txt file and ***NewFile*** variable will be file descriptor of ***target.txt*** file. When we call *copy()* function in *main()*, ***OldFile*** variable (i.e. file descriptor of ***source.txt***) will go as "***old***" variable in *copy()* function and ***NewFile*** variable (i.e. file descriptor of ***target.txt***) will go as "***new***" variable in *copy()* function.

Now coming back our discussion of process...

The executable program ***copy.o*** will contain following things:
1) A set of *headers*, which will describe attributes of this file.
2) The machine instruction form of our program's code.
3) The machine language representation of program's data (i.e. variables in the program), and instructions for the *Kernel* of Operating System for how much memory should *Kernel* store for these variables.
4) Other sections, like Symbol Table, etc.

If we consider our program ***copy.o*** then...
- ✓ The second content, i.e. machine instructions will be the object code of the text of *main()* & *copy()* functions.
- ✓ The third content, i.e. the program's data will be global data, one un-initialized i.e. ***char buffer [2048]*** and one initialized i.e. ***int TempGlobalData = 1***.
- ✓ The first & fourth contents are system dependant and not program dependant, hence not mentioned.

Now consider another program ***copycall.o***, as its name suggests, it is going to call ***copy.o*** inside it. The code will be something like follows:

```
/* Program calling copy.o inside it - name copycall.o */
# include <stdio.h>

void main(int argc, char *argv [1] )
{
        /* Command line argument number error checking */
        if ( argc != 3 )
        {
                printf ( "Invalid Number Of Arguments\n" ) ;
                exit (1) ;
        }
        if ( fork == 0 )
                execl ( "copy", "copy", argv [1], argv [2], 0 ) ;
        wait ( ( int * ) 0 ) ;
        printf ( "Copy Done !\n" ) ;
}
```

Here ***copycall.o*** program calls ***copy.o*** program by *fork()* system call. Obviously, ***copycall.o*** program's process will be the *Parent Process* and when it starts ***copy.o*** in it (i.e. after *fork()*), then ***copy.o*** program's process will become *Child Process* of the ***copycall.o*** process.

Whenever user executes a program from command line (i.e. writing program's name & its arguments and then pressing ***ENTER*** key), *Kernel* load that program in RAM and now called as *Process* (not a program any more).

This load process (i.e. program in RAM) must have at least 3 parts... (Also called as ***Regions***)
1) ***Text -*** i.e. the machine language code of your program's code.
2) ***Data -*** i.e. memory blocks reserved for both initialized & un-initialized global data.
   Stack - explained further.

The ***Kernel creates the Stack dynamically at runtime***. The *Stack* (*i.e. Data structure resembling Stack of coins*) consists of ***frames*** (*imagine one frame as one coin*). When a function is called frame of that function is pushed into the *Stack* and when the same function returns (i.e. finishes), its frame is popped from the *Stack*. In CPU, there is a ***register*** called as "***Stack pointer***"

which continuously keeps the track of the program's depth and can give us the current depth (i.e. how many frames are present now) of the *Stack*.

Each frame resembles each function in the program and thus consists of...

1) Function's parameters } Data part of function
2) Function's local variables }
3) Data necessary to recover the previous *Stack* frame.
   This consists of: ! Value of program's counter
   ! Value of *Stack* pointer register

The "***Text***" part, which is actually the code of program, contains the machine instructions to maintain the growth of the *Stack* and the *Kernel* allocates memory to this growing *Stack* (or de-allocates memory of shrinking *Stack*) as necessary.

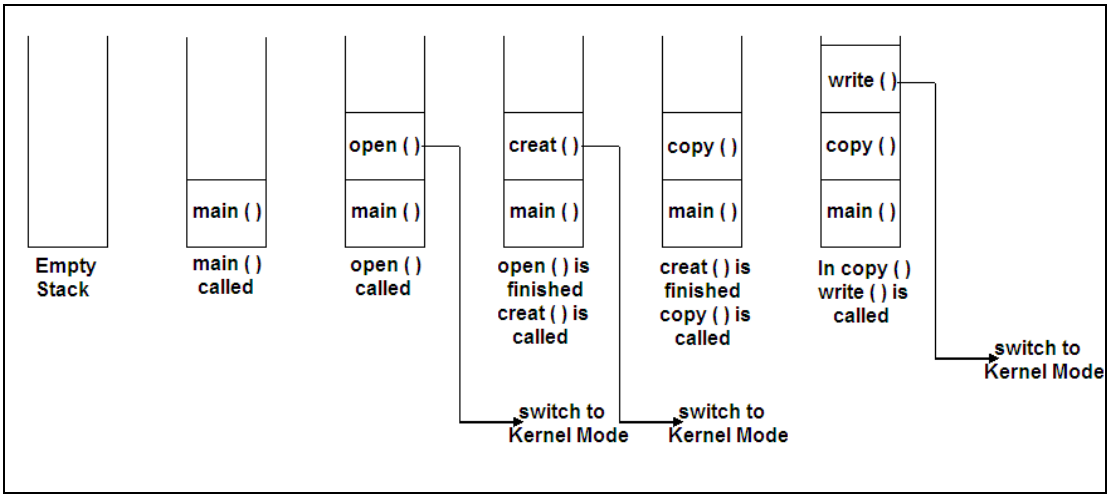
Thus when *main()* function of ***copy.o*** will be called, the *Stack* of ***copy.o***, will get added by logical frame of *main()* of ***copy.o***. And in this frame ***argc***, ***argv [ ]***, ***OldFile***, ***NewFile*** variables will appear. But note that as yet *copy()* is not called, its frame will not appear in *Stack* (and obviously variables of *copy()* function i.e. ***old***, ***new***, ***count*** will also not appear in *Stack*). When program flows further, and *copy()* will be called, then its frame will appear (with appearance of all variables of *copy()* function including its parameters) on the top of the frame of *main()*. When processing of *copy()* function completes and it returns to *main()*, then frame of *copy()* function is removed (popped) from *Stack* and now only frame of *main()* remains on the *Stack*. When *main()* also finishes and returns to its calls program (i.e. to ***copycall.o***) the frame of *main()* is also removed (popped) from the *Stack* and *Stack* becomes empty.

Recall, every process can run in two modes. :

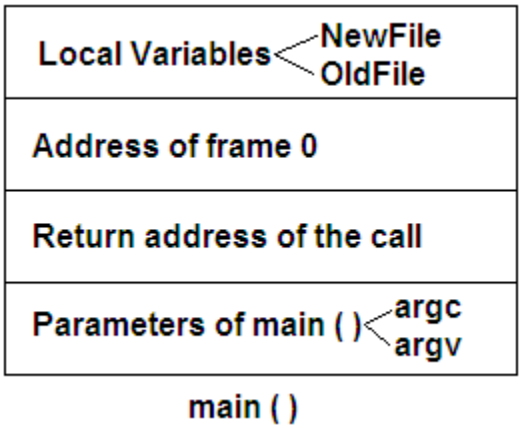
- ***User Mode:*** When *main()* & ***UDF***s are called.
- ***Kernel Mode*:** When system functions are called.

Thus UNIX system keeps 2 independent Stacks for these two modes, for every single process, and called as ***User Stack*** *&* ***Kernel Stack*** *respectively.*

The User Stack: The frames in user *Stack* contain function arguments, local variables and other. When a process starts executing, it reaches from *main()* function. At this point the *Stack* is empty. When *main()* is found (we will take example of copy program) its frame is pushed onto the *Stack* and frame is popped when function returns.



The *User Stack* frame (a single frame) structure is something like follows...



Within this frame 2nd & 3rd contents require some additional description.
The return address of after main() call ! This is the address from *main()* is called, in other words, this is the address to which the caller of *main()* should go after completing the *main()*.
Address of frame 0! This is the address of *frame 0*, which is required to maintain the *Stack*'s growth and shrinkage. The *frame 0* will be of starter process which starts main*()*.

Similar to above frame, when a user defined *copy()* function is called in *main()*. The *Stack* will contain 2 frames, the frame of *main()* and the frame of *copy()*. The frame of *copy()* will contain its parameters ***old*** & ***new***. Then it will contain the return address to which it should go after finishing *copy()* task, then it will contain address of *frame 1* (obviously, address of *main()*'s frame) and lastly it will contain local variables in *copy()* i.e. ***count***.
Thus frame of *main()* will be *frame 1*, frame *copy()* will be *frame 2* and so on....

The Kernel Stack : The program makes use of 4 system calls, *open()*, *creat()*, *read()* & *write()*. Whenever a system call is arrived, the process leaves the *User Stack* (i.e. leaves the *User Mode*) and enters into *Kernel Stack* (i.e. enters in *Kernel Mode*). Thus at beginning, when there is no system call, the *Kernel Stack* is empty, When a system call is made, *Kernel* links it to system call library (shown in figure) where there is entry point (like *main()*) of that system call. The system call library is written entirely in *Assembly Language*. When *Kernel* reaches the system call library, it takes entry point of that respective system call and enters into body of it. Inside system call's body, there are special "***trap instructions***". When executed, these instructions generate "***interrupt***" and here, there is hardware switch to *Kernel Mode*. The call to system call function is same like call to any other function and hence frames of this function (and other system call functions) are then added to *Kernel Stack*. The pushing and popping is also same as that of *User Stack*. But in *User Stack* we know the frames (because we know the functions) which is not possible in *Kernel Stack* because we only know the name of system call function. We don't know whether this system call is calling some other another system calls inside it or not. Hence if we diagrammatically want to show the growth or shrink of *Kernel Stack* we have to use obituary names like *func1*, *func2*, etc...

Data Structures Of a Process

1) u area : -

When a process gets executed, it puts its entry in *Kernel*'s process table. When *Kernel* realizes this it allocates a block of memory called as ***u table*** for this running process. (*u area* is also called *u block*, where "*u*" stands for "*user*"). u *area* contains private data usable only by *Kernel*. Thus *Kernel* can directly access the *u area* of a particular process. But can not access *u area* of another process. Then a question may arise *how multi-processing achieved by Kernel?*

Kernel has a reference (pointer) to a structure variable called as *u*. *Kernel* uses this "*pointer to u*" to access "*u area*" of currently running process. When another process is executed, *Kernel* re-arranges its virtual address spaces in such a way that "*pointer to u*" will not point to "*u area*" of new process. Thus we can say, "*u area contains information only of currently running process*". Contents of *u area* are:

a) A pointer to the process table slot of the currently running process.
b) Parameters of the current system call, return value of current system call and error codes of current system call.
c) File descriptor for all open files.
d) Internal I/O parameters
e) Current directory and current root.
f) Process size limit and file size limit.

2) Process Table: -

This table contains pointer to "***per-process region table***" . *Kernel* can access particular process's region table pointer directly from a pointer to *u area*. In other words, suppose a *process A* is running. Then its entry is made into this process table (which is an array of such entry point of all running process). Then *Kernel* creates *u area* for this *process A* and using *u area*, it finds out *pointer to u* structure variable and by this pointer, it can find out the *A*'s slot in process table. Fields in the process table are...

a) A static field.
b) Identifies of all users who own this running process (***UID***s)
c) An "event descriptor set" if a running process is suspended in sleep state.

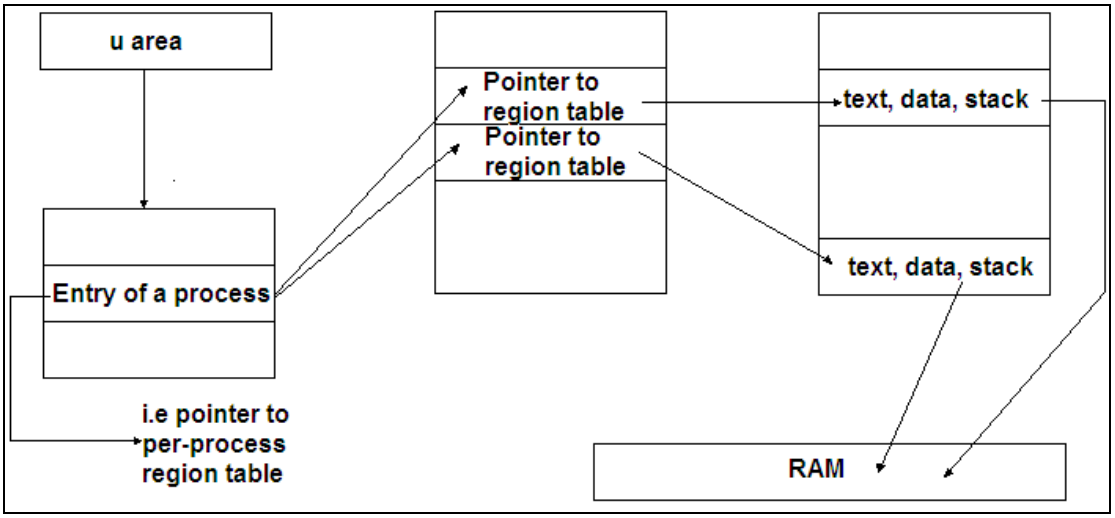
3) Per-process Region Table: -

This contains the pointer to "*region table*" of a process. *Kernel* accesses this table (via *pointer to u* structure variable as explained above), to find out "*region*" of a process in region table.

4) Region Table: -

Region table is a contiguous area (array) of process's address space. (We know that process's address space is nothing but text, data & *Stack* of that process). Thus an entry of a process in region table tells *Kernel* many things like, *whether process contains text or data*, *whether the process is private or shared*, *where the "data" of the process is located in RAM*, etc.

We might surprise to see that, process table should....directly communicate with this region table. *Then why the per-process region table is necessary?*



The answer is that, the extra level of indirection (i.e. use of extra pointer of per-process region table) allows two independent (different) processes to share regions of same process.

What actually Kernel does with the region table?

- ✓ When a running process (like *copycall.o*) calls *exec()* to start another process, the *Kernel* allocates regions (in region table) for "*text*", "*data*" & "s*tack*" of new process. But if region table is full, it frees "*data*", "*text*" & "s*tack*" regions of old process.
- ✓ When a running process calls *fork()*, the *Kernel* duplicates the address space of the old process so sharing becomes possible or otherwise by duplication it really creates new copy of the old process in region table and also in RAM.
- ✓ When a running process calls *exit()*, the *Kernel* frees the using regions from the region table.

Though all above description of process's data structure look complicated, their meanings & relations will become clearer as we go further.

B) Context Of A Process

It is a package of process related information. Its contents are….

a) State of process (i.e. whether running or waiting or sleep)
b) Values of global variables in the process.
c) Values of CPU registers, which are used by the process.
d) Values in its process table slot.
e) Its *u area*.
f) Contents of its *User Stack* (if any)
g) Contents of its *Kernel Stack* (if any)

We know that Operating System is itself a program. Means it itself has its own "*text*", "*data*" & "*stack*". All processes share these parts and hence obviously they are not part of context of a process.

As "*Context of process*" contains virtually all information about the process, we said, "*When a process is running, it is running in the context of that process*".

When another process is to be executed, *Kernel* does "***context switch***" (i.e. change over), means it leaves the context of current process and enters into the context of new process.

This does not mean that *Kernel* can do "*context switch*" any time. Rather *Kernel* does "*context switch*" only under specific conditions (discussed later).

Now when a *Parent Process* calls a *Child Process*, *Kernel* does "*context switch*" from parent to child. But this does not mean that it leaves parent's context as it is. Instead, while leaving a process *Kernel* saves some vital information of the leaving process, so that when it will come back to parent, it can start *Parent Process* as if nothing is happened. Further execution starts at the point where it was left during *context switch*.

We can apply this same strategy for a singe process when it runs in two modes. i.e. *User* & *Kernel* M*odes*. While jumping from one mode to another, *Kernel* saves vital information of leaving mode so that when it comes back, it can be able to execute the previous mode as if nothing is happened. And similarly, further execution will start from where it was left off.

Though above similarity is there, "*The switching of mode*" and "*switching of context*" are two entirely different entities. Simply "*switching of mode occurs in a single process*" while "*switching of context occurs in multiple processes*."

If you see the figure (Chapter 1), "*context switch*" is from A to B while "*mode switch*" is from K to U of A.

Though the code of process does not have any interrupt in it, *Kernel* itself can cause interrupt in running process. In such cases same above "*switching*" is done and by saving vital information *Kernel* can start the process after process's interrupt. ***Note that***, interrupt is not a different process started (i.e. spawned) by *Kernel*. Rather interrupt is hardware dependent.

C) Process States

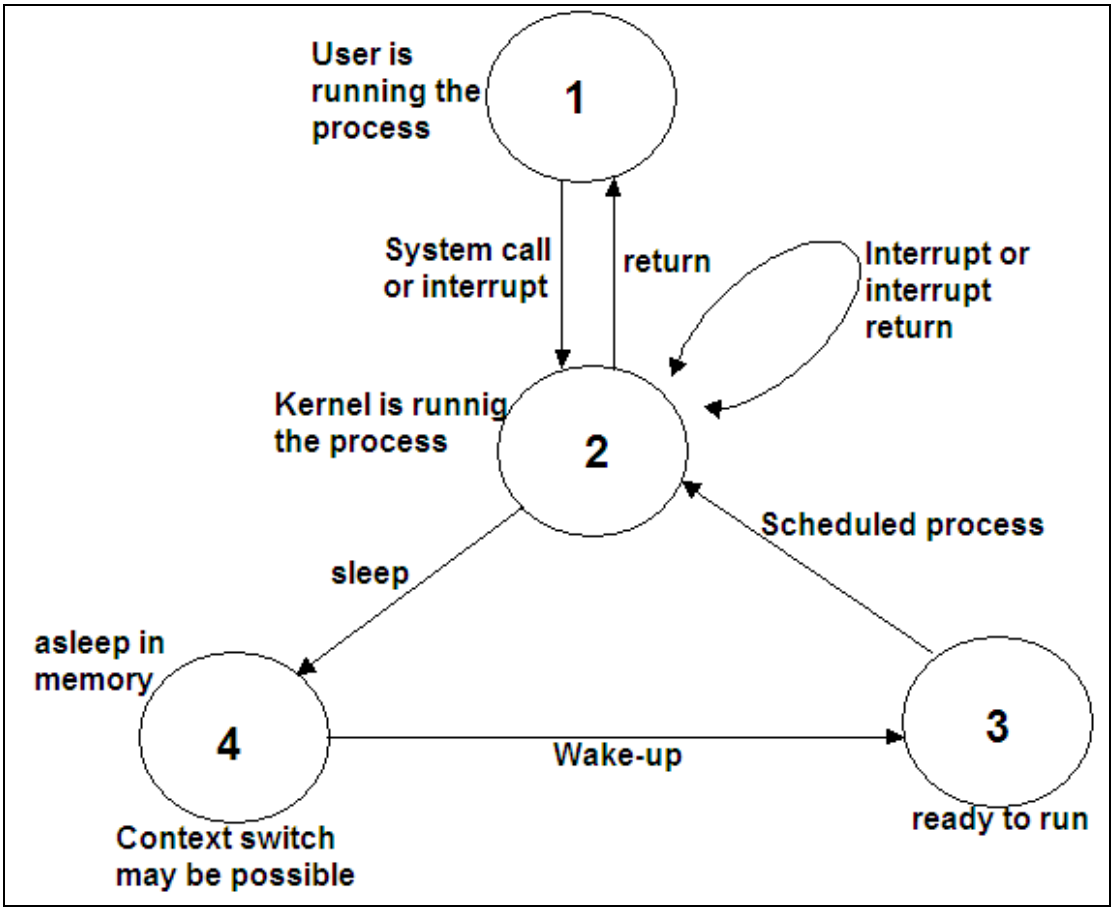
The lifetime of a process can be sub-divided into set of states called as "***Process States***". We are going to see all process states in much detail later on. But here we should know some of them...

a) The process is currently running in *User Mode*.
b) The process is currently running in *Kernel Mode*.
c) The process is not running right now, but it is ready to run when CPU scheduler chooses it. Many processes may be in this state and the CPU scheduling algorithm decides which should be executed next.
d) The process is sleeping. A process can keep in sleeping state, when it can no longer be kept executing because it may be waiting for completion of some I/O.

Though fast mechanism of CPU scheduling gives us an illusion of multiple processes is running simultaneously, actually they are not. Really speaking, CPU can execute one process at a time. Thus at most one process can be in state A & B, which are nothing but two modes of a process.

D) Process State Transitions

CPU scheduling algorithm keeps processes move continuously between above states.



Process States & Transitions

A state transition diagram (as shown above) is a directed graph (refer to graph data structure). ***Nodes*** (i.e. circles in above figure) represent ***states of process*** while ***edges*** (i.e. arrows in figure) represent the events, which force process to move from one state to another (i.e. from one circle to

another circle in figure). ***Note that*** a process will follow one and only transition at a time, which depends upon current system event. State transitions are legal or valid between two states (i.e. between two circles) if and only if, there exists an edge (i.e. arrows) from the first state to second state (i.e. from first circle to second circle)

Kernel itself is a program and it has its own global data, which is shared by processes. In *UNIX* like multi-processing system several processes can execute simultaneously in a time-shared manner as given by CPU scheduling. Now all running processes may be running in their own *Kernel Mode*. Thus all are sharing the *Kernel*'s global data. So it is quite possible that conflict between running processes while accessing *Kernel* data may result in ***corruption of Kernel's global data structure***. To avoid this *Kernel* keep: -

a) Constraints on obituary mode switch & *context switch*.
b) Controls the occurrence of interrupts.

! To allow safe switching, *Kernel* allow *context switch*, only when a process moves from the state of a "*Kernel is running the process*" (i.e. circle 2) to the state of "*asleep in memory*" (i.e. circle 4).

! So process running in *Kernel Mode* (i.e. now *Kernel* is running the process) cannot be preempted by another process. Thus many times it is said, "***Though UNIX system is preemptive, its Kernel itself is non-preemptive***". This is because *Kernel* does not allow another process to preempt a process, which is right now in *Kernel Mode*. But system can keep preempting of multiple processes when they are in *User Mode*.

By keeping itself non-preemptive, Kernel keeps its global data structure's consistency.

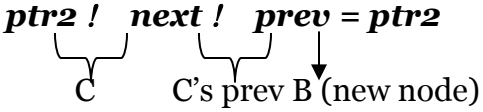
To get above idea more clear, we will take an example. Consider the following code…

```
#include <stdio.h>

struct queue
{
        int data ;
        struct queue * next ;
        struct queue * prev ;
} ;
void main (void)
{
        struct queue *ptr1, *ptr2 ;
        …..
        ….
        ….// Code like memory allocation, etc.
        ….
        ptr2 !    next = ptr1 !    next ;          !    1)
        ptr2 !    prev = ptr1 ;                    !    2)
        ptr1 !    next = ptr2 ;                    !    3)
        /* consider that context switch occurs here */
        /* statement which should execute after context switch is…. */
        ptr2 !    next !    prev = ptr2 ;          !    4)
        ….
        ….
        ….
}
```

First we will examine the code…

- ✓ Here assume that initial coding made ***ptr1*** & ***ptr2*** valid pointers. Now recall what we do to add a node "in-between place" in *doubly linked list*….
- ✓ First we reach that position after which we want to add the node. Assume that initial code already done here.
- ✓ Then we connect the addable node's forward pointer to the forward pointer of the node after which we are going to add the addable node.
- ✓ So if ***ptr1*** is the node after which we are going to add ***ptr2*** and ***ptr2*** is the node that we want to add, then the statement will be…
  ptr2 ! next = ptr1 ! next = ptr2 ;
  So if want to add node B between A & C, first we know that A's next is now C, but after B, B's next will be C (which is nothing but A's next right now).
- ✓ By above statement we took care of linking of new node with the further list. But this is a doubly linked list, hence we yet don't make any connection with new node and the backward list.
  To do this, we will connect new node's backward pointer to the "cut end" node of the list. The "cut end" is that where we add new node. So cut end node of the list is ***ptr1*** and now we are going to join new node's (i.e. ***ptr2***) backward pointer to ***ptr1***.
  Means after establishing B's *next* with C, now we shall connect B's *prev* with A. This is done in above statement.
- ✓ So new node is added. And its connections with back & front nodes are established. But still the "cut end" node (i.e. back node i.e. ***ptr1***) is not connected with new node (i.e. ***ptr2***)
  Note that the "cut end" node ***ptr1*** is in connection with the list. So there is no worry about ***ptr1***'s backward pointer. But as we add ***ptr2*** between ***ptr1*** & some other node (i.e. previously ***ptr1! next***) ***ptr1***'s further connection is lost, which is made by….
  ptr1 ! next = ptr2 ;
  Now ***ptr2*** will be the next node of ***ptr1***. So within A, B, C here we make A's next connected to newly arrived B.
- ✓ Our duty was not to teach data structure. But for clear understanding we explained the code above. Now imagine that *context switch* occurred here (means some other processes executed, blocking the flow of current process).
  Actually process was ready to execute next statement like follows….
- ✓ Connection of B's prev & next completed. Connection A's next with B is completed (as there is no worry about A's prev). Now question remains about C, here C is connected with further "cut list". So no worry about C's next. But due to cutting of A----C and adding B in between A & C, C's prev is lost, which should be connected with B. We may say C's prev = B, but after cutting the list, as A is connected with header node "A" remains in memory at known location. But C gets cut from header and hence though it remains in memory its location is lost somewhere for us. So we should find out C in terms of B. Now we know B is connected with C and A is connected with B. So C is nothing but B's next's ***ptr1! next*** is C. We want to assign C's backward pointer to B, hence…



After studying the code, now come back to our process of *context switch*…

If *context switch* occurs at given position in above code, all forward linking is correctly four… but as last statement is not executed (due to *context switch*), the backward linking is incorrect. Now if some another process tries to use this link list, the list will be destroyed completely.

UNIX system does not allow above situation to occur. Means it will not allow some another process to execute (i.e. turning attention of *Kernel* to new process while *Kernel* is already executing the old one) while *Kernel* is executing some other process in *Kernel Mode*.

So when a new process is executed and *Kernel* gets its indication, *Kernel* first completes ***critical section*** of code of currently running process, then current process goes sleep state and then makes "*context switch"* to deal with the newly coming process.

Similar to above problem (i.e. arrival of new process while old process is already running in *Kernel Mode*), there is yet another problem. Consider that this time, not a new process, but an interrupt occurs!

For confliction of processes, it was quite easy to keep awaiting a new process until the critical code section of old process is completed. But as interrupt has higher priority, if they are coming from hardware or network then *Kernel* should deal with them first. If it continues dealing with old process, system performance of handling interrupt diminishes and if it deals with interrupts, the linked list (of previous example) will get corrupted.

For such "*To be or not to be*" situation, *Kernel* raises priority of the process on "***Process Execution Level***", stops the interrupt for a while, completes the critical section of code of process and then immediately handles the interrupt.

Our intention was to tell capabilities of *Kernel* in above situation. But such situations are quite rare!

E) Sleep and Wake-up

Above discussion may mislead us that *Kernel* is capable of pushing some process forcibly to fell a sleep, No! Not at all! Processes are quiet independent and thus take their own decisions while going in any state transition. But process is designed in such a way and with such rules (we will see them later) that, if situation like "above critical code section", after completing such section, the process, having such code section can go to sleep if some other process or interrupt is waiting for *Kernel*. But again note that "following the rules" is process's self-initiative not a burden by someone else!

Processes go to sleep because they are waiting for a particular even to occur. Such events may be waiting for I/O completion, waiting for exit, waiting for availability of some system resource, etc. Thus it is usually said that "***Process sleeps on an event***", means the process will remain slept until the event does not occurred. When event occurs, process wake up and enter into the state of "*Ready to run*" (circle 2 in figure).

Many processes can go to sleep at a time waiting for an event to occur. As soon as the event occurs, all slept processes wake-up and enter into state of "*ready to run*". Here these processes become eligible for CPU scheduling and thus not executed immediately but get executed when CPU schedules them. Sleep process does not consume any CPU resource. Also *Kernel* does not keep tracking of slept process. Instead, when event occurs, it awakes the slept process.

We will take one example. Suppose there is memory space called buffer. When a process uses this buffer, it is locked so that no other process at same time should able to access the same buffer.

So when *process A*, at the very beginning accesses the buffer (because at the very beginning we assume that buffer is unlocked), now process enter in *Kernel Mode* and locks the buffer taking care if it sleeps at some time later on.

The *Kernel* implements such a lock as follows :...

```
while ( lock condition is true)
{
        sleep ( event : the condition becomes false ) ;
}
again set condition to true ;
```

So *Kernel* unlocks the buffer and wake-up all processes (which are at "***asleep***" state, circle 4 in figure) by something like following code...

set condition to false ;
wakeup(event : the condition becomes false) ;

Now consider 3 process A, B & C which all want to use the same buffer. They all want to sleep whenever buffer is locked and will wake-up when buffer is unlocked.

One process at a time, finds the buffer, looks for whether buffer is locked or not, if locked, goes to sleep waiting for the event of unlocking the buffer.

Eventually, sometimes, the buffer gets unlocked and all the processes wake-up (if slept before) and enter into the state of "*ready to run*". On the current CPU scheduling possibility, *Kernel* chooses one of the waken-up process, say B to start its execution. Now B executes above *while loop*, finds that buffer is unlocked, uses it as per its needs, then locks it again and proceeds further by setting the lock condition to true.

Now suppose B goes to sleep on waiting for some other event like I/O and thus *Kernel* chooses one of the remaining process (A or C) to execute, then process A enters its *while loop*, finds that buffer is locked and again goes to sleep. Process C may do the same thing. Now the I/O event for which B is waiting, occurs and thus B awakens, and unlocks the buffer, this unlocking event wakes up A & C, and *Kernel* chooses one of them (according to CPU scheduling) to execute which now can access the unlocked buffer.

So, we can say that the "while – sleep()" method allows only one process to access the system resource. (In above example, system resource is memory space – i.e. buffer). A process enters the sleep state instantaneously and stays in slept state until it wakes-up. After sleeping of this process *Kernel* chooses another process to run and switches its context to the context of that new process.

III) Kernel Data Structures :

Usually *Kernel* data structures are global, so as to allow access to multiple processes. But, though global, *Kernel* keeps consistency of its data structures by allowing context switching only during "*sleep stage of processes*" and by rearranging "*process execution level"* for interrupts.

The global data structures of *Kernel* are usually "*tables*" (arrays) of fixed size rather than dynamic allocated size. The good effect of "fixed size" ... *Kernel* operation simple but the bad effect is that entries in global data structure become limited (...to fixed size)

System administrator usually gives this fixed size at the time of system installation. So if this size overflows, the system flags error. On the other hand, if user of this system does not use system to its full extent, the allocated fixed sizes may look too bit as if there is wasting of memory.

But note that system designers believe much in simplicity and hence most of the *Kernel* code is array based and simple loop based rather than some complicated and error prone dynamic allocation.

Super User & Common User With Respect To Kernel :

In *UNIX*, system administrator is considered as "*Super user*" (common user also can become *super user* after fulfilling some tasks) while all others are "*common users*". So when a program/command is executed by a super user & by a common user, the kernel does not distinguish them as separate. The distinguishing features between the two are only concerned with file permissions & privileges. With respect to *Kernel* processes for both are of equal privilege.