B.Tech. (Seventh Semester) Examination 2013

Advance Operating System (IT4101)

(Information Technology)

Model Answer

....

Section A

Q.1 - **(10×1=10)**

1. `ls -l` command is used for :

a. List of files
b. List of log files
c. Long list
d. Directories

Ans: C. Long list

2. "AWK" is derived from the initials of the language's three developers: _________, ________, and _________. (Fill in the blanks.)

Ans: Aho, Weinberger, Kernighan.

3. `PWD` in UNIX stands for _______________.

Ans: Present working directory.

4. Wc -l command is used for :

a. Count word length
b. Count number of lines
c. Count characters
d. Count words

Ans: B. Count number of lines.

5. Pipe is a type of ______ file.

Ans: Special file.

6. ifree algorithm is used for releasing disk blocks. [True / False]

Ans: False

7. The two __________ addresses represent the u-area of two processes, but the Kernel access them via the same _________ address. [Fill in the blanks]

Ans: Physical, Virtual

8. The buffer pool follows the _______ algorithm.

Ans: LRU

9. The Algorithm to free a disk block is:

a. Ifree
b. Brelse
c. free
d. delete

Ans: C. Free

10. The two types of operating systems for distributed system are ___________and ____________.
Ans: Tightly coupled and loosely coupled

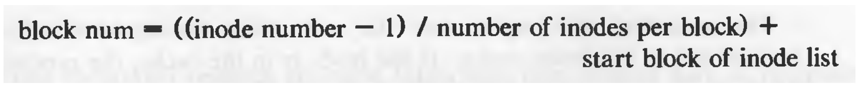
Q.2 - (5×2=10)

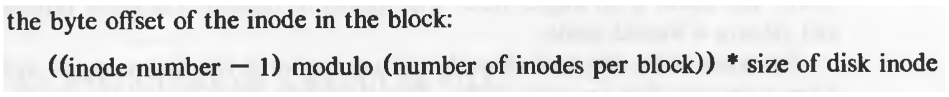
1. What is the kernel of an operating system?
Ans: Proper definition of kernel

2. What is a shell in operating system?
ANS: Proper definition of Shell

3. Write the formula to compute the block number that contains the particular inode, also calculate byte offset.

ANS:

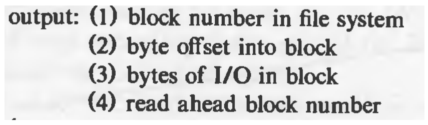

block num = ((inode number − 1) / number of inodes per block) + start block of inode list


the byte offset of the inode in the block:

((inode number − 1) modulo (number of inodes per block)) * size of disk inode

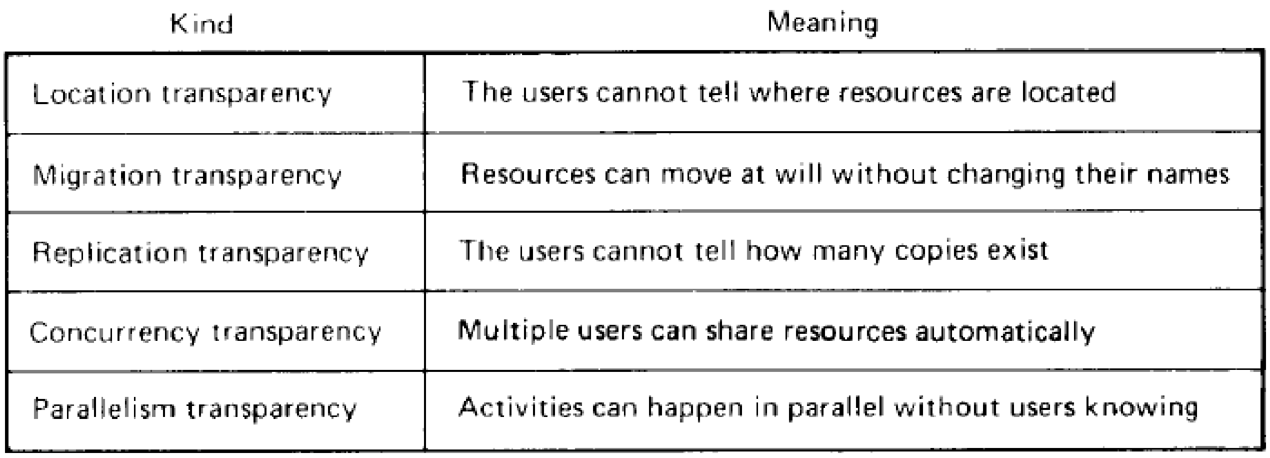
4. What is the output of bmap algorithm?

ANS:


```
output: (1) block number in file system
        (2) byte offset into block
        (3) bytes of I/O in block
        (4) read ahead block number
```

5. What are the different types of transparency in a distributed system?

ANS:



Kind	Meaning
Location transparency	The users cannot tell where resources are located
Migration transparency	Resources can move at will without changing their names
Replication transparency	The users cannot tell how many copies exist
Concurrency transparency	Multiple users can share resources automatically
Parallelism transparency	Activities can happen in parallel without users knowing

Fig. 1-13. Different kinds of transparency in a distributed system.

Section B

Note: Attempt any one question from each unit. Each question carries 8 marks.

UNIT 1

Q 1. Explain the use of array in AWK with proper example.

ANS: Write following points:

1. About AWK
2. Declaration of Arrays in AWK
3. Use of Arrays in AWK
4. Example

Q 2. Describe standard I/O and Pipe as the building block primitives of UNIX with examples.

ANS:

As described earlier, the philosophy of the UNIX system is to provide operating system primitives that enable users to write small, modular programs that can be used as building blocks to build more complex programs. One such primitive visible to shell users is the capability to *redirect I/O*. Processes conventionally have access to three files: they read from their *standard input* file, write to their *standard output* file, and write error messages to their *standard error* file. Processes executing at a terminal typically use the terminal for these three files, but each may be "redirected" independently. For instance, the command line

```
ls
```

lists all files in the current directory on the standard output, but the command line

```
ls > output
```

redirects the standard output to the file called "output" in the current directory, using the *creat* system call mentioned above. Similarly, the command line

```
mail mjb < letter
```

*open*s the file "letter" for its standard intput and *mail*s its contents to the user named "mjb." Processes can redirect input and output simultaneously, as in

```
nroff −mm < doc1 > doc1.out 2> errors
```

where the text formatter *nroff* reads the input file *doc1*, redirects its standard output to the file *doc1.out*, and redirects error messages to the file *errors* (the notation "2>" means to redirect the output for file descriptor 2, conventionally the standard error). The programs *ls*, *mail*, and *nroff* do not know what file their standard input, standard output, or standard error will be; the shell recognizes the symbols "<", ">", and "2>" and sets up the standard input, standard output, and standard error appropriately before executing the processes.

The second building block primitive is the *pipe*, a mechanism that allows a stream of data to be passed between reader and writer processes. Processes can redirect their standard output to a pipe to be read by other processes that have redirected their standard input to come from the pipe. The data that the first processes write into the pipe is the input for the second processes. The second processes could also redirect their output, and so on, depending on programming need. Again, the processes need not know what type of file their standard output is; they work regardless of whether their standard output is a regular file, a pipe, or a device. When using the smaller programs as building blocks for a larger, more complex program, the programmer uses the pipe primitive and redirection of I/O to integrate the piece parts. Indeed, the system tacitly encourages such programming style so that new programs can work with existing programs.

For example, the program *grep* searches a set of files (parameters to grep) for a given pattern:

```
grep main a.c b.c c.c
```

searches the three files a.c, b.c, and c.c for lines containing the string "main" and prints the lines that it finds onto standard output. Sample output may be:

```
a.c: main(argc, argv)
c.c: /* here is the main loop in the program */
c.c: main()
```

The program *wc* with the option −l counts the number of lines in the standard input file. The command line

```
grep main a.c b.c c.c | wc −l
```

counts the number of lines in the files that contain the string "main"; the output from *grep* is "piped" directly into the *wc* command. For the previous sample output from *grep*, the output from the piped command is

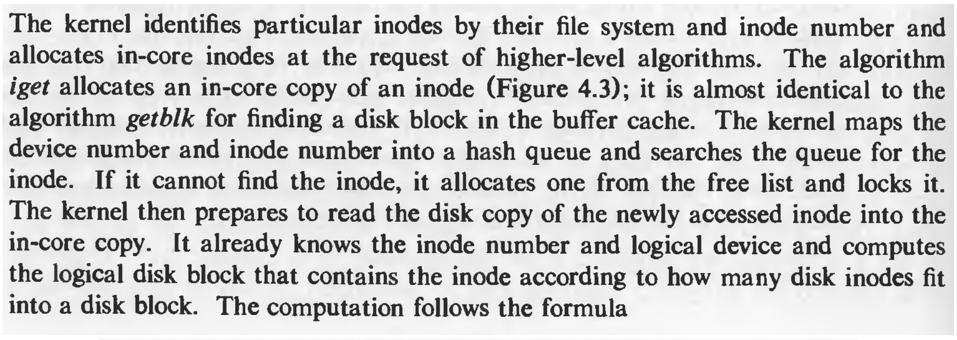
```
3
```

The use of pipes frequently makes it unnecessary to create temporary files.

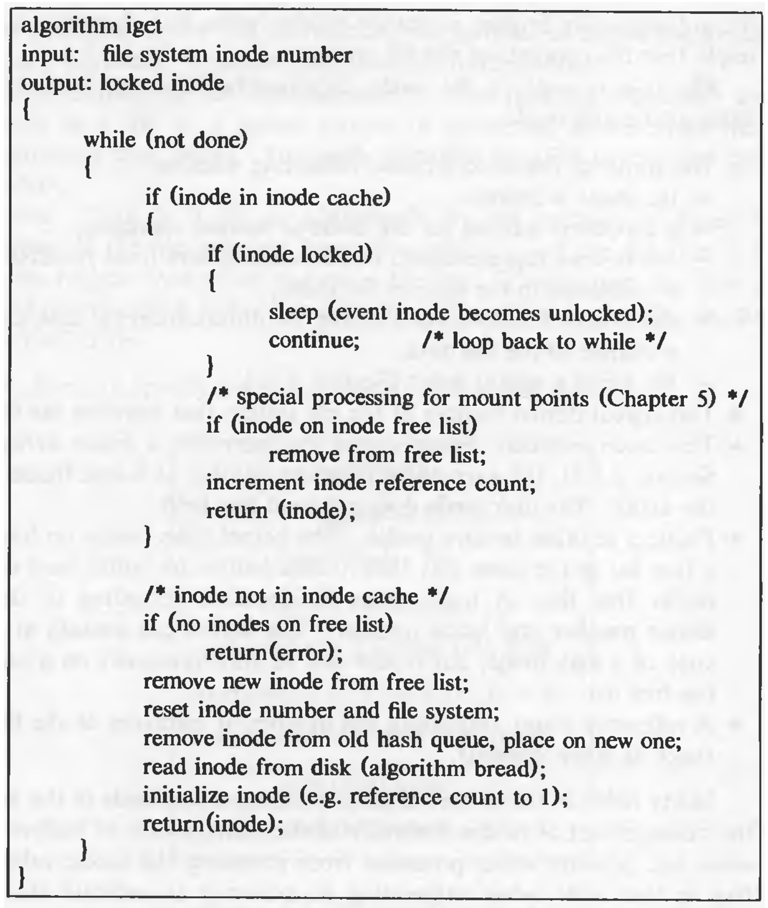
UNIT 2

Q 3. Write and explain the algorithm for allocation of incore inodes.

ANS:



The kernel identifies particular inodes by their file system and inode number and allocates in-core inodes at the request of higher-level algorithms. The algorithm *iget* allocates an in-core copy of an inode (Figure 4.3); it is almost identical to the algorithm *getblk* for finding a disk block in the buffer cache. The kernel maps the device number and inode number into a hash queue and searches the queue for the inode. If it cannot find the inode, it allocates one from the free list and locks it. The kernel then prepares to read the disk copy of the newly accessed inode into the in-core copy. It already knows the inode number and logical device and computes the logical disk block that contains the inode according to how many disk inodes fit into a disk block. The computation follows the formula



```
algorithm iget
input:  file system inode number
output: locked inode
{
    while (not done)
    {
        if (inode in inode cache)
        {
            if (inode locked)
            {
                sleep (event inode becomes unlocked);
                continue;       /* loop back to while */
            }
            /* special processing for mount points (Chapter 5) */
            if (inode on inode free list)
                remove from free list;
            increment inode reference count;
            return (inode);
        }

        /* inode not in inode cache */
        if (no inodes on free list)
            return(error);
        remove new inode from free list;
        reset inode number and file system;
        remove inode from old hash queue, place on new one;
        read inode from disk (algorithm bread);
        initialize inode (e.g. reference count to 1);
        return(inode);
    }
}
```

Figure 4.3. Algorithm for Allocation of In-Core Inodes

block num = ((inode number − 1) / number of inodes per block) + start block of inode list

where the division operation returns the integer part of the quotient. For example, assuming that block 2 is the beginning of the inode list and that there are 8 inodes per block, then inode number 8 is in disk block 2, and inode number 9 is in disk block 3. If there are 16 inodes in a disk block, then inode numbers 8 and 9 are in disk block 2, and inode number 17 is the first inode in disk block 3.

When the kernel knows the device and disk block number, it reads the block using the algorithm *bread* (Chapter 2), then uses the following formula to compute the byte offset of the inode in the block:

((inode number − 1) modulo (number of inodes per block)) * size of disk inode

For example, if each disk inode occupies 64 bytes and there are 8 inodes per disk block, then inode number 8 starts at byte offset 448 in the disk block. The kernel removes the in-core inode from the free list, places it on the correct hash queue, and sets its in-core reference count to 1. It copies the file type, owner fields, permission settings, link count, file size, and the table of contents from the disk inode to the in-core inode, and returns a locked inode.

The kernel manipulates the inode lock and reference count independently. The lock is set during execution of a system call to prevent other processes from accessing the inode while it is in use (and possibly inconsistent). The kernel releases the lock at the conclusion of the system call: an inode is never locked across system calls. The kernel increments the reference count for every active reference to a file. For example, Section 5.1 will show that it increments the inode reference count when a process *open*s a file. It decrements the reference count only when the reference becomes inactive, for example, when a process *close*s a file. The reference count thus remains set across multiple system calls. The lock is free between system calls to allow processes to share simultaneous access to a file; the reference count remains set between system calls to prevent the kernel from reallocating an active in-core inode. Thus, the kernel can lock and unlock an allocated inode independent of the value of the reference count. System calls other than *open* allocate and release inodes, as will be seen in Chapter 5.

Returning to algorithm *iget*, if the kernel attempts to take an inode from the free list but finds the free list empty, it reports an error. This is different from the philosophy the kernel follows for disk buffers, where a process sleeps until a buffer becomes free: Processes have control over the allocation of inodes at user level via execution of *open* and *close* system calls, and consequently the kernel cannot guarantee when an inode will become available. Therefore, a process that goes to sleep waiting for a free inode to become available may never wake up. Rather than leave such a process "hanging," the kernel fails the system call. However, processes do not have such control over buffers: Because a process cannot keep a buffer locked across system calls, the kernel can guarantee that a buffer will become free soon, and a process therefore sleeps until one is available.

The preceding paragraphs cover the case where the kernel allocated an inode that was not in the inode cache. If the inode is in the cache, the process (A) would find it on its hash queue and check if the inode was currently locked by another process (B). If the inode is locked, process A sleeps, setting a flag in the in-core inode to indicate that it is waiting for the inode to become free. When process B later unlocks the inode, it awakens all processes (including process A) waiting for the inode to become free. When process A is finally able to use the inode, it locks the inode so that other processes cannot allocate it. If the reference count was previously 0, the inode also appears on the free list, so the kernel removes it from there: the inode is no longer free. The kernel increments the inode reference count and returns a locked inode.

Q 4. Write and explain the algorithm to release an incore inode.

ANS:

4.1.3 Releasing Inodes

When the kernel releases an inode (algorithm *iput*, Figure 4.4), it decrements its in-core reference count. If the count drops to 0, the kernel writes the inode to disk if the in-core copy differs from the disk copy. They differ if the file data has changed, if the file access time has changed, or if the file owner or access permissions have changed. The kernel places the inode on the free list of inodes, effectively caching the inode in case it is needed again soon. The kernel may also release all data blocks associated with the file and free the inode if the number of links to the file is 0.

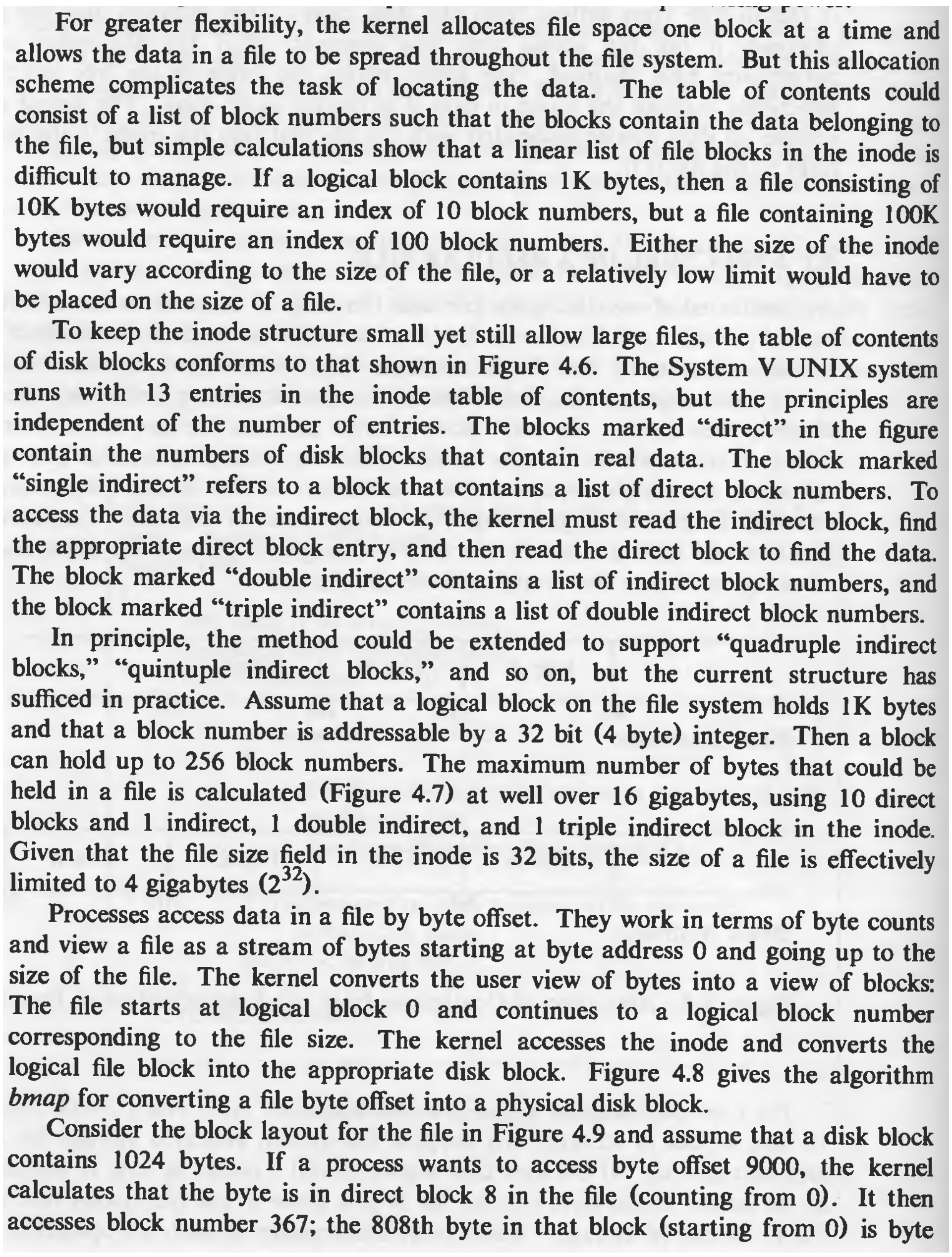
```
algorithm iput                    /* release (put) access to in—core inode */
input:  pointer to in—core inode
output: none
{
     lock inode if not already locked;
     decrement inode reference count;
     if (reference count == 0)
     {
          if (inode link count == 0)
          {
               free disk blocks for file (algorithm free, section 4.7);
               set file type to 0;
               free inode (algorithm ifree, section 4.6);
          }
          if (file accessed or inode changed or file changed)
               update disk inode;
          put inode on free list;
     }
     release inode lock;
}
```

Figure 4.4. Releasing an Inode

UNIT 3

Q 5. How disk block addresses are managed in inode? If a process wants to access byte offset 992640124 in a file, find out the desired block and byte offset in block for process.

ANS:



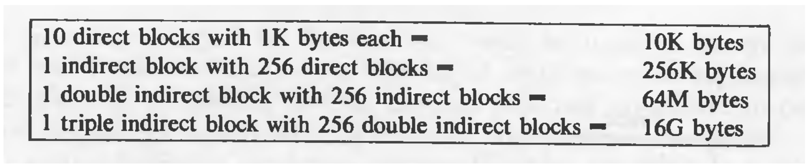
For greater flexibility, the kernel allocates file space one block at a time and allows the data in a file to be spread throughout the file system. But this allocation scheme complicates the task of locating the data. The table of contents could consist of a list of block numbers such that the blocks contain the data belonging to the file, but simple calculations show that a linear list of file blocks in the inode is difficult to manage. If a logical block contains 1K bytes, then a file consisting of 10K bytes would require an index of 10 block numbers, but a file containing 100K bytes would require an index of 100 block numbers. Either the size of the inode would vary according to the size of the file, or a relatively low limit would have to be placed on the size of a file.

To keep the inode structure small yet still allow large files, the table of contents of disk blocks conforms to that shown in Figure 4.6. The System V UNIX system runs with 13 entries in the inode table of contents, but the principles are independent of the number of entries. The blocks marked "direct" in the figure contain the numbers of disk blocks that contain real data. The block marked "single indirect" refers to a block that contains a list of direct block numbers. To access the data via the indirect block, the kernel must read the indirect block, find the appropriate direct block entry, and then read the direct block to find the data. The block marked "double indirect" contains a list of indirect block numbers, and the block marked "triple indirect" contains a list of double indirect block numbers.

In principle, the method could be extended to support "quadruple indirect blocks," "quintuple indirect blocks," and so on, but the current structure has sufficed in practice. Assume that a logical block on the file system holds 1K bytes and that a block number is addressable by a 32 bit (4 byte) integer. Then a block can hold up to 256 block numbers. The maximum number of bytes that could be held in a file is calculated (Figure 4.7) at well over 16 gigabytes, using 10 direct blocks and 1 indirect, 1 double indirect, and 1 triple indirect block in the inode. Given that the file size field in the inode is 32 bits, the size of a file is effectively limited to 4 gigabytes (2^{32}).

Processes access data in a file by byte offset. They work in terms of byte counts and view a file as a stream of bytes starting at byte address 0 and going up to the size of the file. The kernel converts the user view of bytes into a view of blocks: The file starts at logical block 0 and continues to a logical block number corresponding to the file size. The kernel accesses the inode and converts the logical file block into the appropriate disk block. Figure 4.8 gives the algorithm *bmap* for converting a file byte offset into a physical disk block.

Consider the block layout for the file in Figure 4.9 and assume that a disk block contains 1024 bytes. If a process wants to access byte offset 9000, the kernel calculates that the byte is in direct block 8 in the file (counting from 0). It then accesses block number 367; the 808th byte in that block (starting from 0) is byte



10 direct blocks with 1K bytes each =	10K bytes
1 indirect block with 256 direct blocks =	256K bytes
1 double indirect block with 256 indirect blocks =	64M bytes
1 triple indirect block with 256 double indirect blocks =	16G bytes

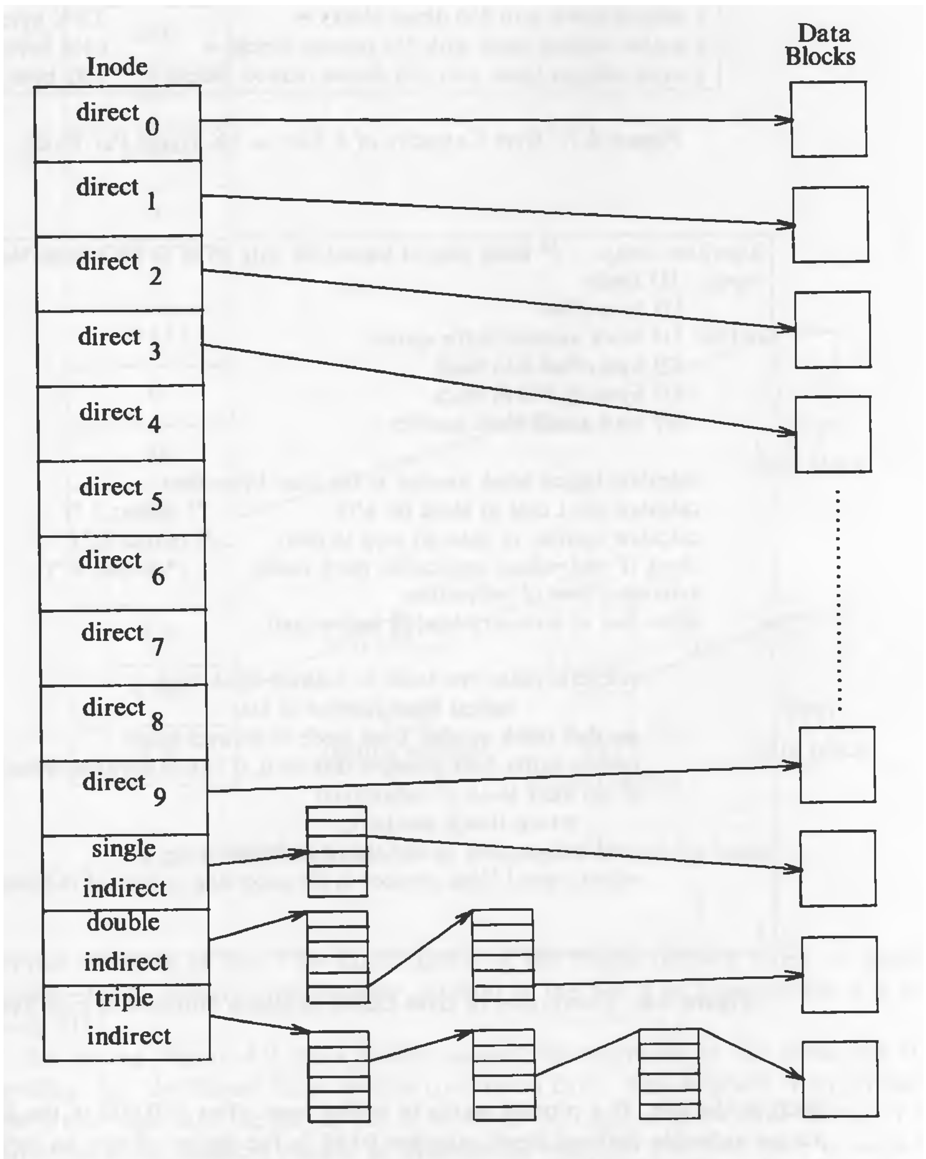
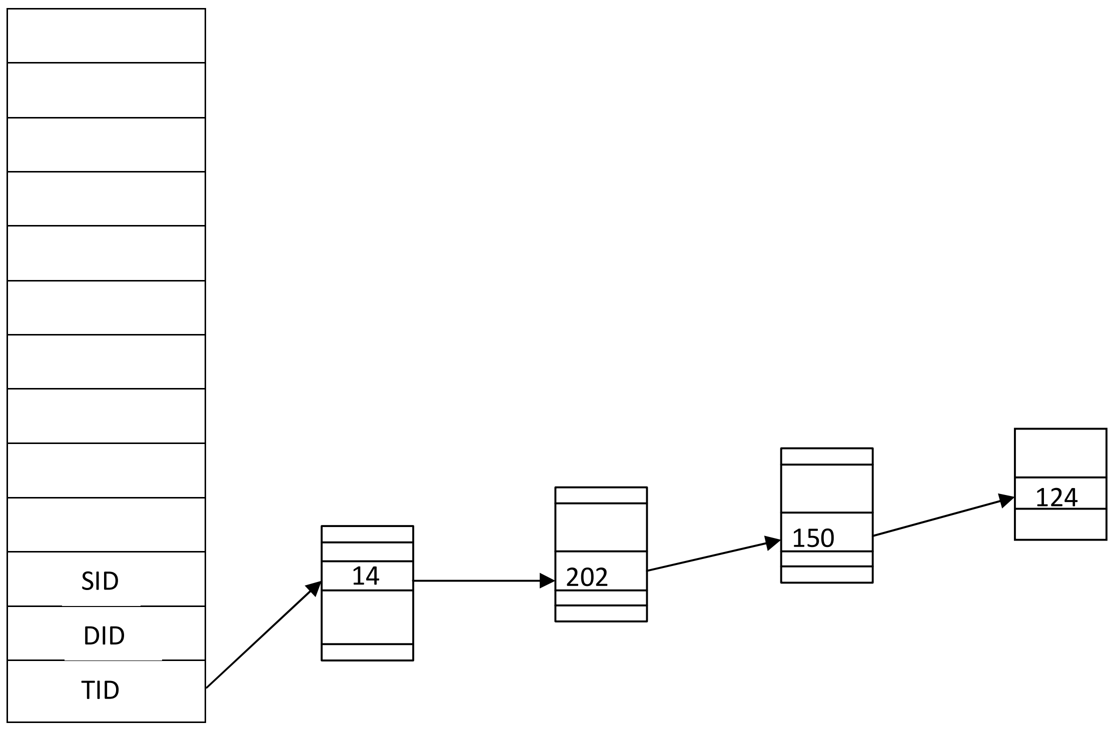


Figure 4.6. Direct and Indirect Blocks in Inode

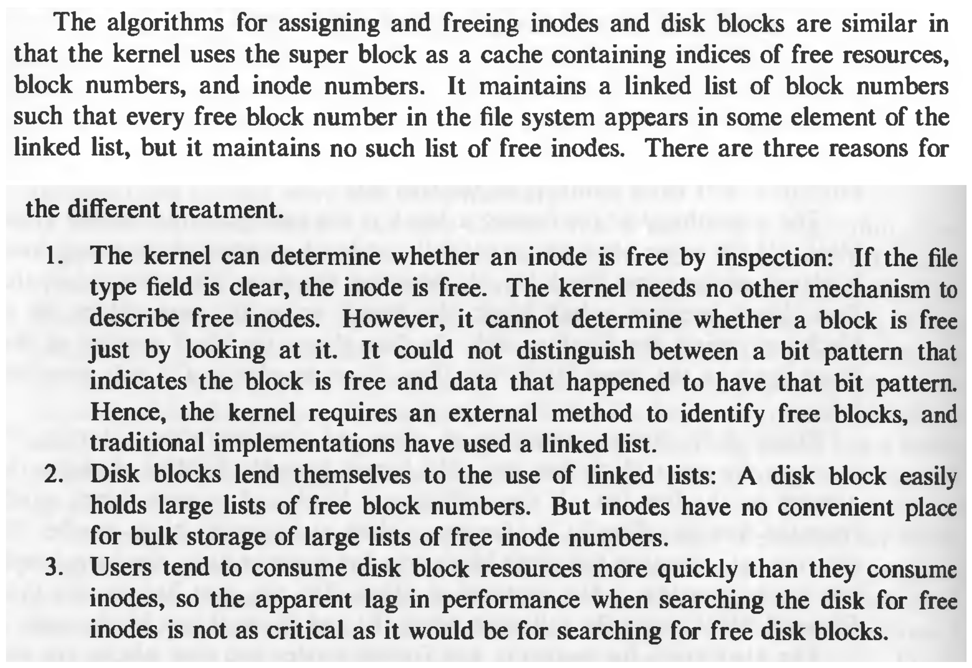
Solution of Problem:

Step 1: 992640124 – 10 x 1024	=	992629884	Direct Blocks
Step 2: 992629884 – 256x 1024	=	992367740	Single Indirect
Step 3: 992367740 – 256x 256x 1024	=	925258876	Double Indirect
Step 4: 925258876 – 256x 256x 256x 1024	=	-16254610308	Triple Indirect
Step 5: 925258876 /(256x 256x 1024)	=	13.787	14th Double Indirect Block
Step 6:925258876 – (13x 256x 256x 1024)	=	52843644	
Step 7: 52843644 / (256x 1024)	=	201.58	202nd Single Indirect Block
Step 8: 52843644 – (201x 256x 1024)	=	152700	
Step 9: 152700 / 1024	=	149.121	150th direct block
Step10: 152700 – 1024x 149	=	124	Byte offset 124 in Block



Q 6. Write and explain the algorithm for freeing of disk block with proper diagrams.

ANS:



The algorithms for assigning and freeing inodes and disk blocks are similar in that the kernel uses the super block as a cache containing indices of free resources, block numbers, and inode numbers. It maintains a linked list of block numbers such that every free block number in the file system appears in some element of the linked list, but it maintains no such list of free inodes. There are three reasons for the different treatment.

1. The kernel can determine whether an inode is free by inspection: If the file type field is clear, the inode is free. The kernel needs no other mechanism to describe free inodes. However, it cannot determine whether a block is free just by looking at it. It could not distinguish between a bit pattern that indicates the block is free and data that happened to have that bit pattern. Hence, the kernel requires an external method to identify free blocks, and traditional implementations have used a linked list.
2. Disk blocks lend themselves to the use of linked lists: A disk block easily holds large lists of free block numbers. But inodes have no convenient place for bulk storage of large lists of free inode numbers.
3. Users tend to consume disk block resources more quickly than they consume inodes, so the apparent lag in performance when searching the disk for free inodes is not as critical as it would be for searching for free disk blocks.

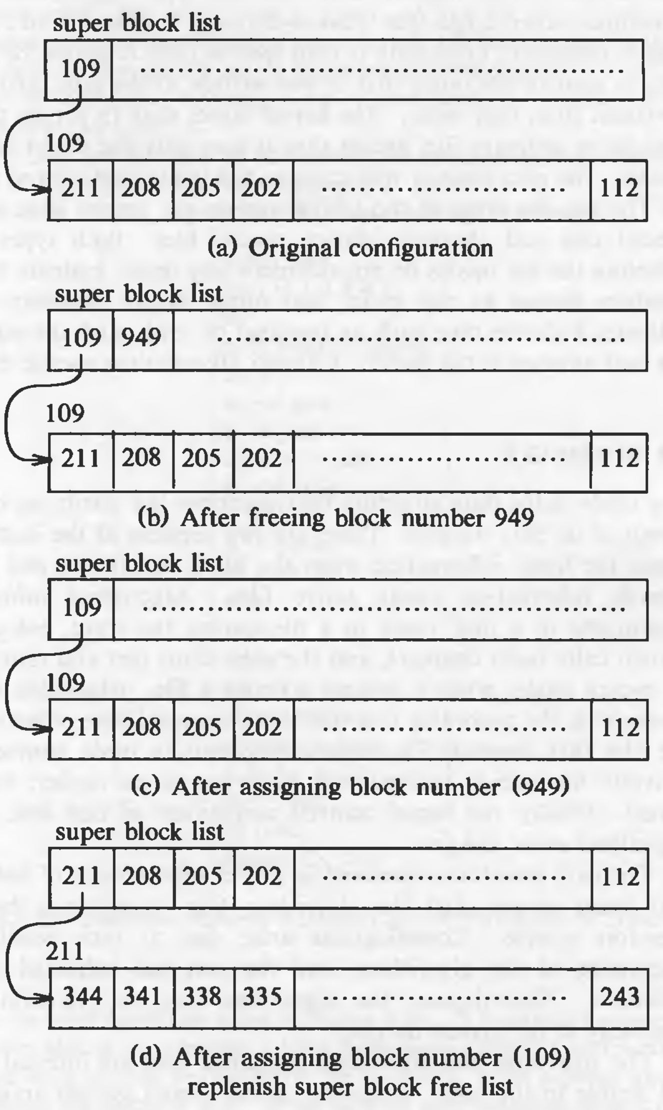


Figure 4.20. Requesting and Freeing Disk Blocks

UNIT 4

Q 7. How the size of a process can be changed? Write and explain the algorithm to change the size of a process.

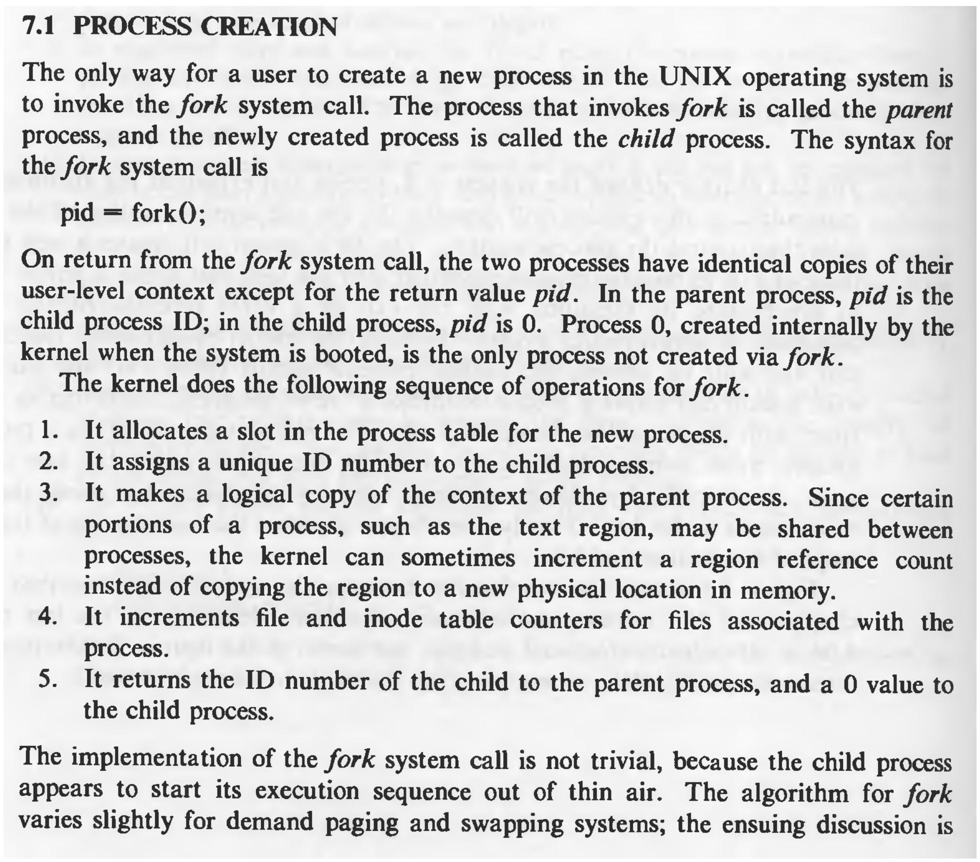
ANS:

Write following points

1. What is a size of process and how it can be changed
2. Write the algorithm to change the size of a process
3. Explain the algorithm with diagram

Q 8. Write and explain the algorithm for ~~sleep~~ Creation of a process.

ANS:



7.1 PROCESS CREATION

The only way for a user to create a new process in the UNIX operating system is to invoke the *fork* system call. The process that invokes *fork* is called the *parent* process, and the newly created process is called the *child* process. The syntax for the *fork* system call is

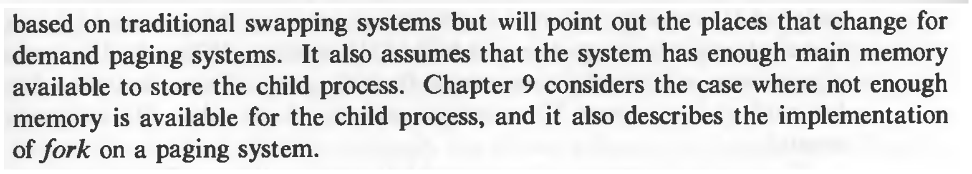
```
pid = fork();
```

On return from the *fork* system call, the two processes have identical copies of their user-level context except for the return value *pid*. In the parent process, *pid* is the child process ID; in the child process, *pid* is 0. Process 0, created internally by the kernel when the system is booted, is the only process not created via *fork*.

The kernel does the following sequence of operations for *fork*.

1. It allocates a slot in the process table for the new process.
2. It assigns a unique ID number to the child process.
3. It makes a logical copy of the context of the parent process. Since certain portions of a process, such as the text region, may be shared between processes, the kernel can sometimes increment a region reference count instead of copying the region to a new physical location in memory.
4. It increments file and inode table counters for files associated with the process.
5. It returns the ID number of the child to the parent process, and a 0 value to the child process.

The implementation of the *fork* system call is not trivial, because the child process appears to start its execution sequence out of thin air. The algorithm for *fork* varies slightly for demand paging and swapping systems; the ensuing discussion is



based on traditional swapping systems but will point out the places that change for demand paging systems. It also assumes that the system has enough main memory available to store the child process. Chapter 9 considers the case where not enough memory is available for the child process, and it also describes the implementation of *fork* on a paging system.

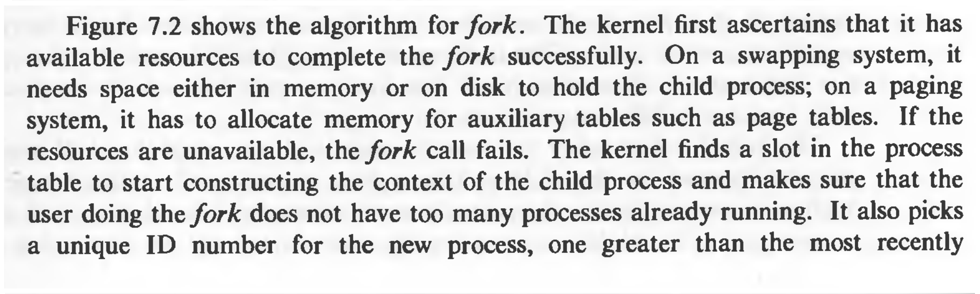
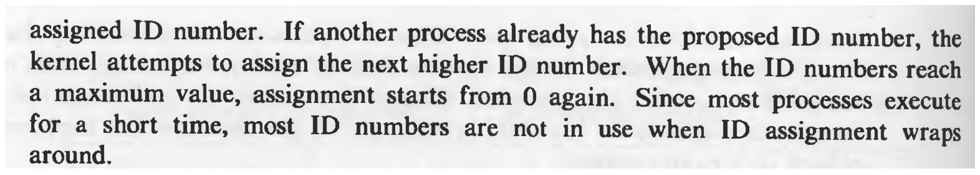


Figure 7.2 shows the algorithm for *fork*. The kernel first ascertains that it has available resources to complete the *fork* successfully. On a swapping system, it needs space either in memory or on disk to hold the child process; on a paging system, it has to allocate memory for auxiliary tables such as page tables. If the resources are unavailable, the *fork* call fails. The kernel finds a slot in the process table to start constructing the context of the child process and makes sure that the user doing the *fork* does not have too many processes already running. It also picks a unique ID number for the new process, one greater than the most recently



assigned ID number. If another process already has the proposed ID number, the kernel attempts to assign the next higher ID number. When the ID numbers reach a maximum value, assignment starts from 0 again. Since most processes execute for a short time, most ID numbers are not in use when ID assignment wraps around.

```
algorithm fork
input:   none
output: to parent process, child PID number
        to child process, 0
{
        check for available kernel resources;
        get free proc table slot, unique PID number;
        check that user not running too many processes;
        mark child state "being created;"
        copy data from parent proc table slot to new child slot;
        increment counts on current directory inode and changed root (if applicable);
        increment open file counts in file table;
        make copy of parent context (u area, text, data, stack) in memory;
        push dummy system level context layer onto child system level context;
                        dummy context contains data allowing child process
                        to recognize itself, and start running from here
                        when scheduled;
        if (executing process is parent process)
        {
                change child state to "ready to run;"
                return(child ID);       /* from system to user */
        }
        else      /* executing process is the child process */
        {
                initialize u area timing fields;
                return(0);        /* to user */
        }
}
```

Figure 7.2. Algorithm for Fork

The kernel next initializes the child's process table slot, copying various fields from the parent slot. For instance, the child "inherits" the parent process real and effective user ID numbers, the parent process group, and the parent *nice* value, used for calculation of scheduling priority. Later sections discuss the meaning of these fields. The kernel assigns the parent process ID field in the child slot, putting the child in the process tree structure, and initializes various scheduling parameters, such as the initial priority value, initial CPU usage, and other timing fields. The initial state of the process is "being created" (recall Figure 6.1).

The kernel now adjusts reference counts for files with which the child process is automatically associated. First, the child process resides in the current directory of the parent process. The number of processes that currently access the directory increases by 1 and, accordingly, the kernel increments its inode reference count. Second, if the parent process or one of its ancestors had ever executed the *chroot* system call to change its root, the child process inherits the changed root and increments its inode reference count. Finally, the kernel searches the parent's user file descriptor table for open files known to the process and increments the global file table reference count associated with each open file. Not only does the child process inherit access rights to open files, but it also shares access to the files with the parent process because both processes manipulate the same file table entries. The effect of *fork* is similar to that of *dup* vis-a-vis open files: A new entry in the user file descriptor table points to the entry in the global file table for the open file. For *dup*, however, the entries in the user file descriptor table are in one process; for *fork*, they are in different processes.

The kernel is now ready to create the user-level context of the child process. It allocates memory for the child process *u area*, regions, and auxiliary page tables, duplicates every region in the parent process using algorithm *dupreg*, and attaches every region to the child process using algorithm *attachreg*. In a swapping system, So far, the kernel has created the static portion of the child context; now it creates the dynamic portion. The kernel copies the parent context layer 1, containing the user saved register context and the kernel stack frame of the *fork* system call. If the implementation is one where the kernel stack is part of the *u area*, the kernel automatically creates the child kernel stack when it creates the child *u area*. Otherwise, the parent process must copy its kernel stack to a private area of memory associated with the child process. In either case, the kernel stacks for the parent and child processes are identical. The kernel then creates a dummy context layer (2) for the child process, containing the saved register context for context layer(1). It sets the program counter and other registers in the saved register context so that it can "restore" the child context, even though it had never executed before, and so that the child process can recognize itself as the child when it runs. For instance, if the kernel code tests the value of register 0 to decide if the process is the parent or the child, it writes the appropriate value in the child saved register context in layer 1. The mechanism is similar to that discussed for a context switch in the previous chapter.

When the child context is ready, the parent completes its part of *fork* by changing the child state to "ready to run (in memory)" and by returning the child process ID to the user. The kernel later schedules the child process for execution via the normal scheduling algorithm, and the child process "completes" *its* part of the *fork*. The context of the child process was set up by the parent process; to the kernel, the child process appears to have awakened after awaiting a resource. The child process executes part of the code for the *fork* system call, according to the program counter that the kernel restored from the saved register context in context layer 2, and returns a 0 from the system call.

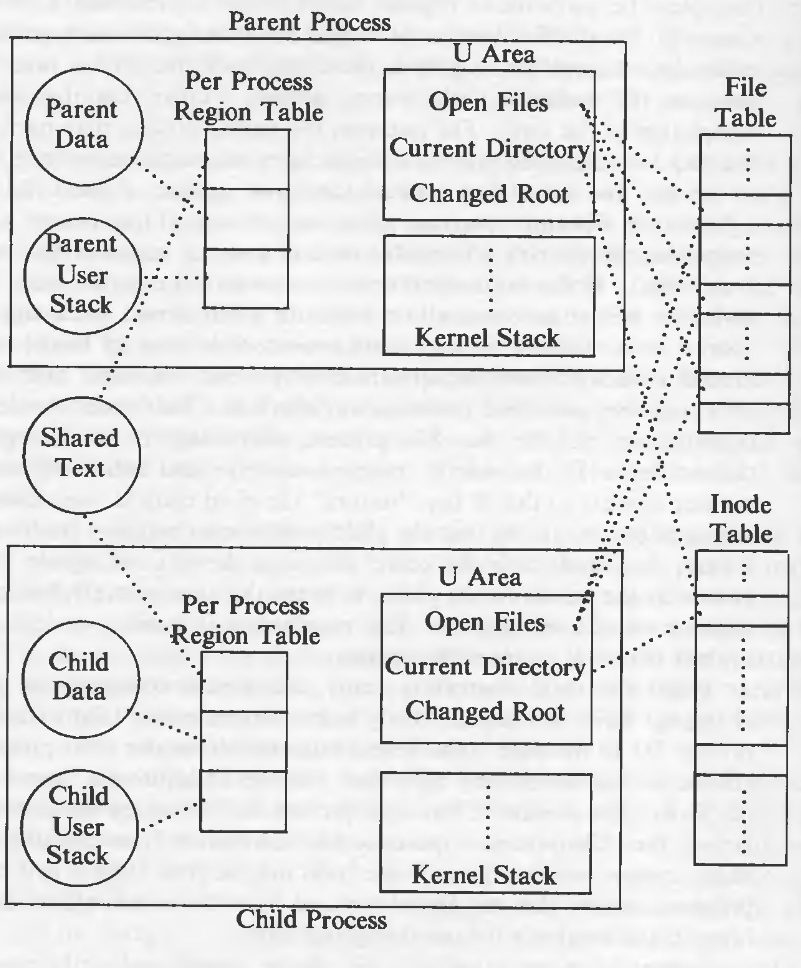


Figure 7.3. Fork Creating a New Process Context

Figure 7.3 gives a logical view of the parent and child processes and their relationship to other kernel data structures immediately after completion of the *fork* system call. To summarize, both processes share files that the parent had open at the time of the *fork*, and the file table reference count for those files is one greater than it had been. Similarly, the child process has the same current directory and changed root (if applicable) as the parent, and the inode reference count of those directories is one greater than it had been. The processes have identical copies of the text, data, and (user) stack regions; the region type and the system implementation determine whether the processes can share a physical copy of the text region.

UNIT 5

Q 9. Explain workstation model of organizing distributed system. Also explain the use of idle workstations.

ANS:

4.2.1. The Workstation Model

The workstation model is straightforward: the system consists of workstations (high-end personal computers) scattered throughout a building or campus and connected by a high-speed LAN, as shown in Fig. 4-10. Some of the workstations may be in offices, and thus implicitly dedicated to a single user, whereas others may be in public areas and have several different users during the course of a day. In both cases, at any instant of time, a workstation either has a single user logged into it, and thus has an "owner" (however temporary), or it is idle.

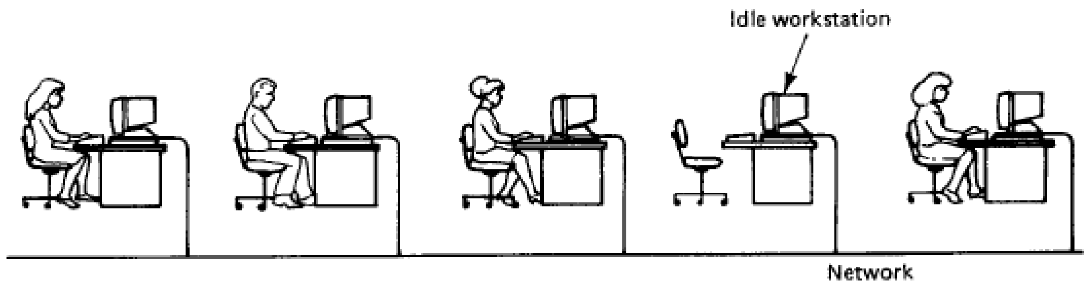


Fig. 4-10. A network of personal workstations, each with a local file system.

In some systems the workstations have local disks and in others they do not. The latter are universally called **diskless workstations**, but the former are variously known as **diskful workstations**, or **disky workstations**, or even stranger names. If the workstations are diskless, the file system must be implemented by one or more remote file servers. Requests to read and write files are sent to a file server, which performs the work and sends back the replies.

Diskless workstations are popular at universities and companies for several reasons, not the least of which is price. Having a large number of workstations equipped with small, slow disks is typically much more expensive than having one or two file servers equipped with huge, fast disks and accessed over the LAN.

A second reason that diskless workstations are popular is their ease of maintenance. When a new release of some program, say a compiler, comes out, the system administrators can easily install it on a small number of file servers in the machine room. Installing it on dozens or hundreds of machines all over a building or campus is another matter entirely. Backup and hardware maintenance is also simpler with one centrally located 5-gigabyte disk than with fifty 100-megabyte disks scattered over the building.

Another point against disks is that they have fans and make noise. Many people find this noise objectionable and do not want it in their office.

Finally, diskless workstations provide symmetry and flexibility. A user can walk up to any workstation in the system and log in. Since all his files are on the file server, one diskless workstation is as good as another. In contrast, when all the files are stored on local disks, using someone else's workstation means that you have easy access to *his* files, but getting to your own requires extra effort, and is certainly different from using your own workstation.

When the workstations have private disks, these disks can be used in one of at least four ways:

1. Paging and temporary files.
2. Paging, temporary files, and system binaries.
3. Paging, temporary files, system binaries, and file caching.
4. Complete local file system.

The first design is based on the observation that while it may be convenient to keep all the user files on the central file servers (to simplify backup and maintenance, etc.) disks are also needed for paging (or swapping) and for temporary files. In this model, the local disks are used only for paging and files that are temporary, unshared, and can be discarded at the end of the login session. For example, most compilers consist of multiple passes, each of which creates a temporary file read by the next pass. When the file has been read once, it is discarded. Local disks are ideal for storing such files.

The second model is a variant of the first one in which the local disks also hold the binary (executable) programs, such as the compilers, text editors, and electronic mail handlers. When one of these programs is invoked, it is fetched from the local disk instead of from a file server, further reducing the network load. Since these programs rarely change, they can be installed on all the local disks and kept there for long periods of time. When a new release of some system program is available, it is essentially broadcast to all machines. However, if that machine happens to be down when the program is sent, it will miss the program and continue to run the old version. Thus some administration is needed to keep track of who has which version of which program.

A third approach to using local disks is to use them as explicit caches (in addition to using them for paging, temporaries, and binaries). In this mode of operation, users can download files from the file servers to their own disks, read and write them locally, and then upload the modified ones at the end of the login session. The goal of this architecture is to keep long-term storage centralized, but reduce network load by keeping files local while they are being used. A disadvantage is keeping the caches consistent. What happens if two users download the same file and then each modifies it in different ways? This problem is not easy to solve, and we will discuss it in detail later in the book.

Fourth, each machine can have its own self-contained file system, with the possibility of mounting or otherwise accessing other machines' file systems. The idea here is that each machine is basically self-contained and that contact with the outside world is limited. This organization provides a uniform and guaranteed response time for the user and puts little load on the network. The disadvantage is that sharing is more difficult, and the resulting system is much closer to a network operating system than to a true transparent distributed operating system.

The one diskless and four diskful models we have discussed are summarized in Fig. 4-11. The progression from top to bottom in the figure is from complete dependence on the file servers to complete independence from them.

The advantages of the workstation model are manifold and clear. The model is certainly easy to understand. Users have a fixed amount of dedicated computing power, and thus guaranteed response time. Sophisticated graphics programs can be very fast, since they can have direct access to the screen. Each user has a large degree of autonomy and can allocate his workstation's resources as he sees fit. Local disks add to this independence, and make it possible to continue working to a lesser or greater degree even in the face of file server crashes.

However, the model also has two problems. First, as processor chips continue to get cheaper, it will soon become economically feasible to give each user first 10 and later 100 CPUs. Having 100 workstations in your office makes it hard to see out the window. Second, much of the time users are not using their workstations, which are idle, while other users may need extra computing capacity and cannot get it. From a system-wide perspective, allocating resources in such a way that some users have resources they do not need while other users need these resources badly is inefficient.

The first problem can be addressed by making each workstation a personal multiprocessor. For example, each window on the screen can have a dedicated CPU to run its programs. Preliminary evidence from some early personal multiprocessors such as the DEC Firefly, suggest, however, that the mean number of CPUs utilized is rarely more than one, since users rarely have more than one active process at once. Again, this is an inefficient use of resources, but as CPUs get cheaper nd cheaper as the technology improves, wasting them will become less of a sin.

4.2.2. Using Idle Workstations

The second problem, idle workstations, has been the subject of considerable research, primarily because many universities have a substantial number of personal workstations, some of which are idle (an idle workstation is the devil's playground?). Measurements show that even at peak periods in the middle of the day, often as many as 30 percent of the workstations are idle at any given moment. In the evening, even more are idle. A variety of schemes have been proposed for using idle or otherwise underutilized workstations (Litzkow et al., the user must tell which machine to use, putting the full burden of keeping track of idle machines on the user. Second, the program executes in the environment of the remote machine, which is usually different from the local environment. Finally, if someone should log into an idle machine on which a remote process is running, the process continues to run and the newly logged-in user either has to accept the lower performance or find another machine.

The research on idle workstations has centered on solving these problems. The key issues are:

1. How is an idle workstation found?
2. How can a remote process be run transparently?
3. What happens if the machine's owner comes back?

Let us consider these three issues, one at a time.

How is an idle workstation found? To start with, what is an idle workstation? At first glance, it might appear that a workstation with no one logged in at the console is an idle workstation, but with modern computer systems things are not always that simple. In many systems, even with no one logged in there may be dozens of processes running, such as clock daemons, mail daemons, news

The algorithms used to locate idle workstations can be divided into two categories: server driven and client driven. In the former, when a workstation goes idle, and thus becomes a potential compute server, it announces its availability. It can do this by entering its name, network address, and properties in a registry file (or data base), for example. Later, when a user wants to execute a

An alternative way for the newly idle workstation to announce the fact that it has become unemployed is to put a broadcast message onto the network. All

Q 10. Describe how RPC is managed with threads in distributed operating system?

ANS:

4.1. THREADS

In most traditional operating systems, each process has an address space and a single thread of control. In fact, that is almost the definition of a process. Nevertheless, there are frequently situations in which it is desirable to have multiple threads of control sharing one address space but running in quasi-parallel, as though they were separate processes (except for the shared address space). In this section we will discuss these situations and their implications.

4.1.1. Introduction to Threads

Consider, for example, a file server that occasionally has to block waiting for the disk. If the server had multiple threads of control, a second thread could run while the first one was sleeping. The net result would be a higher throughput and better performance. It is not possible to achieve this goal by creating two independent server processes because they must share a common buffer cache, which requires them to be in the same address space. Thus a new mechanism is needed, one that historically was not found in single-processor operating systems.

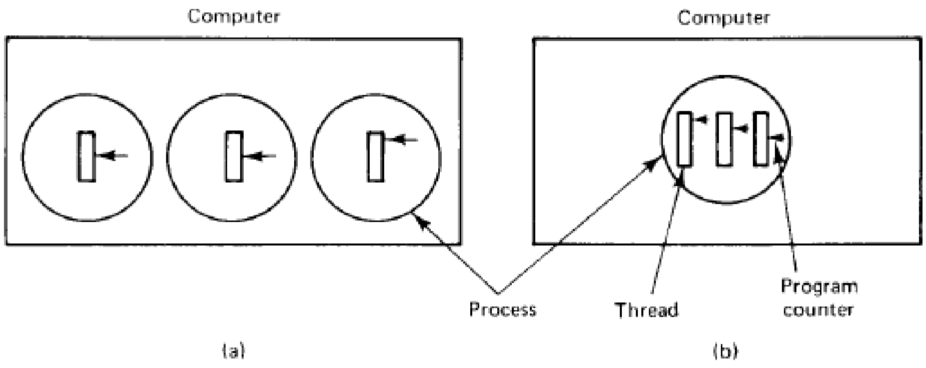


Fig. 4-1. (a) Three processes with one thread each. (b) One process with three threads.

In Fig. 4-1(a) we see a machine with three processes. Each process has its own program counter, its own stack, its own register set, and its own address space. The processes have nothing to do with each other, except that they may be able to communicate through the system's interprocess communication primitives, such as semaphores, monitors, or messages. In Fig. 4-1(b) we see another machine, with one process. Only this process contains multiple threads of control, usually just called **threads**, or sometimes **lightweight processes**. In many respects, threads are like little mini-processes. Each thread runs strictly sequentially and has its own program counter and stack to keep track of where it is. Threads share the CPU just as processes do: first one thread runs, then another does (timesharing). Only on a multiprocessor do they actually run in parallel. Threads can create child threads and can block waiting for system calls to complete, just like regular processes. While one thread is blocked, another thread in

Threads were invented to allow parallelism to be combined with sequential execution and blocking system calls. Consider our file server example again. One possible organization is shown in Fig. 4-3(a). Here one thread, the

dispatcher, reads incoming requests for work from the system mailbox. After examining the request, it chooses an idle (i.e., blocked) **worker thread** and hands it the request, possibly by writing a pointer to the message into a special word associated with each thread. The dispatcher then wakes up the sleeping worker (e.g., by doing an UP on the semaphore on which it is sleeping).

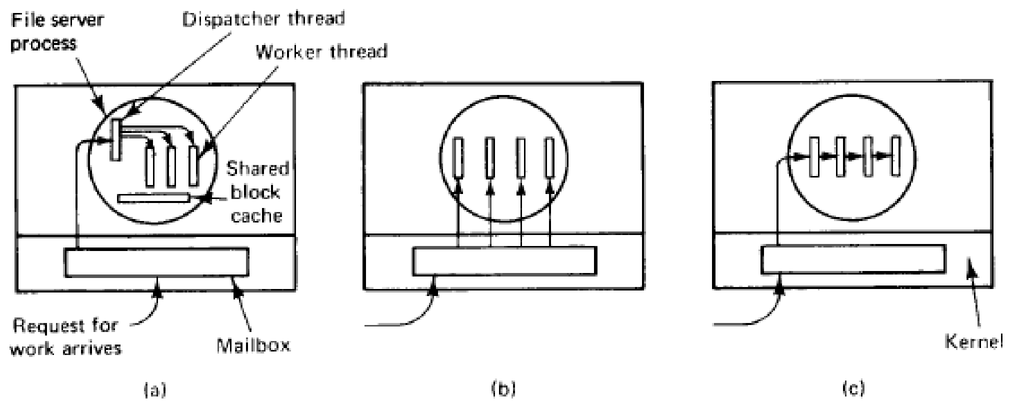


Fig. 4-3. Three organizations of threads in a process. (a) Dispatcher/worker model. (b) Team model. (c) Pipeline model.

When the worker wakes up, it checks to see if the request can be satisfied from the shared block cache, to which all threads have access. If not, it sends a message to the disk to get the needed block (assuming it is a READ) and goes to sleep awaiting completion of the disk operation. The scheduler will now be invoked and another thread will be started, possibly the dispatcher, in order to acquire more work, or possibly another worker that is now ready to run.

Threads can also be organized in the **pipeline** model of Fig. 4-3(c). In this model, the first thread generates some data and passes them on to the next thread for processing. The data continue from thread to thread, with processing going on at each step. Although this is not appropriate for file servers, for other

problems, such as the producer-consumer, it may be a good choice. Pipelining is widely used in many areas of computer systems, from the internal structure of RISC CPUs to UNIX command lines.

Threads are frequently also useful for clients. For example, if a client wants a file to be replicated on multiple servers, it can have one thread talk to each server. Another use for client threads is to handle signals, such as interrupts from the keyboard (DEL or BREAK). Instead of letting the signal interrupt the process, one thread is dedicated full time to waiting for signals. Normally, it is blocked, but when a signal comes in, it wakes up and processes the signal. Thus using threads can eliminate the need for user-level interrupts.

4.1.5. Threads and RPC

It is common for distributed systems to use both RPC and threads. Since threads were invented as a cheap alternative to standard (heavyweight) processes, it is natural that researchers would take a closer look at RPC in this context, to see if it could be made more lightweight as well. In this section we will discuss some interesting work in this area.

Bershad et al. (1990) have observed that even in a distributed system, a substantial number of RPCs are to processes on the same machine as the caller (e.g., to the window manager). Obviously, this result depends on the system, but it is common enough to be worth considering. They have proposed a new scheme that makes it possible for a thread in one process to call a thread in another process on the same machine much more efficiently than the usual way.

The idea works like this. When a server thread, *S*, starts up, it exports its interface by telling the kernel about it. The interface defines which procedures are callable, what their parameters are, and so on. When a client thread *C* starts up, it imports the interface from the kernel and is given a special identifier to use for the call. The kernel now knows that *C* is going to call *S* later, and creates special data structures to prepare for the call.

When a new message comes in to the server's machine, the kernel creates a new thread on-the-fly to service the request. Furthermore, it maps the message into the server's address space, and sets up the new thread's stack to access the message. This scheme is sometimes called **implicit receive** and it is in contrast to a conventional thread making a system call to receive a message. The thread that is created spontaneously to handle an incoming RPC is occasionally referred to as a **pop-up thread**. The idea is illustrated in Fig. 4-9.

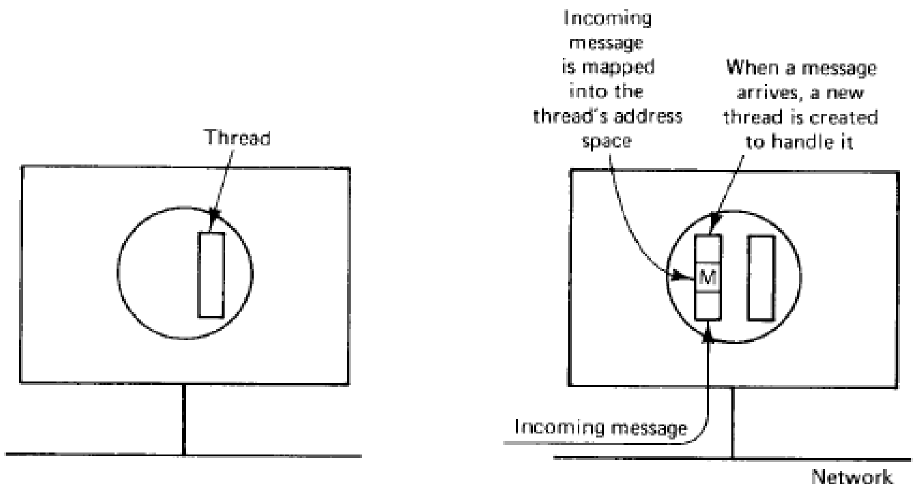

Thread
Incoming message is mapped into the thread's address space
When a message arrives, a new thread is created to handle it
M
Incoming message
Network

Fig. 4-9. Creating a thread when a message arrives.