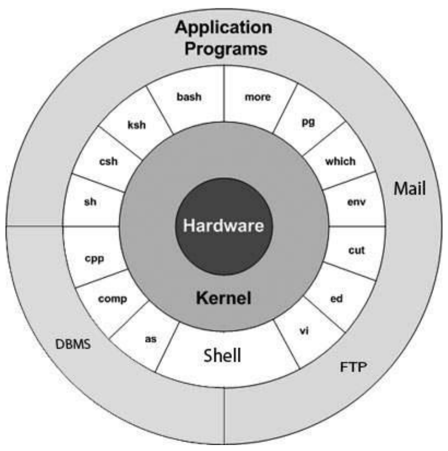
1.Introduction to UNIX / Linux Kernel

1.Explain the Architecture of the UNIX.

Unix has a graphical user interface similar to the Windows operating system that makes it easy for navigation and a good supportive environment. The internal design view of this operating system can be known from its architecture.



1.Hardware: Hardware is the most simple and least powerful layer in the Unix Architecture. Hardware is the components that are humanly visible. Whatever hardware is connected to a Unix operating system-based machine, comes in the hardware layer.

2.**Kernel:** This is the most powerful layer of the Unix architecture. The kernel is responsible for acting as an interface between the user and the hardware for the effective utilization of the hardware. The kernel handles the hardware effectively by using the device drivers. The kernel is also responsible for process management. So, the main 2 features of the kernel are process management and file management.

1.**Process Management:** The processes that execute within the operating system require a lot of management in terms of memory being allocated to them, the resource allocation to the process, process synchronization, etc. All this is done by the Kernel in Unix OS. This is done using various Operating System Techniques like paging, framing, virtual memory, swapping, context-switching, etc.

2.File Management: File management involves managing the data stored in the files. This also includes the transmission of data stored in these files to the processes as and when they request it.
3.Shell: We understood the importance of the kernel and that it handles most of the important and complex tasks of Unix OS. Since the kernel is such an important program of the Unix Operating System, its direct access to the users can be dangerous. Hence, the Shell comes into the picture. Shell is an interpreter program that interprets the commands entered by the user and then sends the requests to the kernel to execute those commands.

4.Applications/Application Programs: The last layer of the Unix architecture is the Application Program layer. As the name suggests, this outermost layer of Unix Architecture is responsible for executing the application programs.

2.Explain the behaviour of following c program.

```
main()

int status;

if(fork()==0)

execk("/bin/date", "date", 0);

wait(&status);
```

Solution

The fork system call in Unix creates a new process. The new process inherits various properties from its parent that is Environmental variables, File descriptors, etc. After a successful fork call, two copies of the original code will be running. In the original process that is the parent, the return value of fork will be the process ID of the child. In the new child process the return value of fork will be 0.When we type "date" on the unix command line, the command line interpreter i.e. "shell" forks o that momentarily 2 shells are running, then the code in the child process is replaced by the sode of the "date" program by using one of the family of exec system calls.

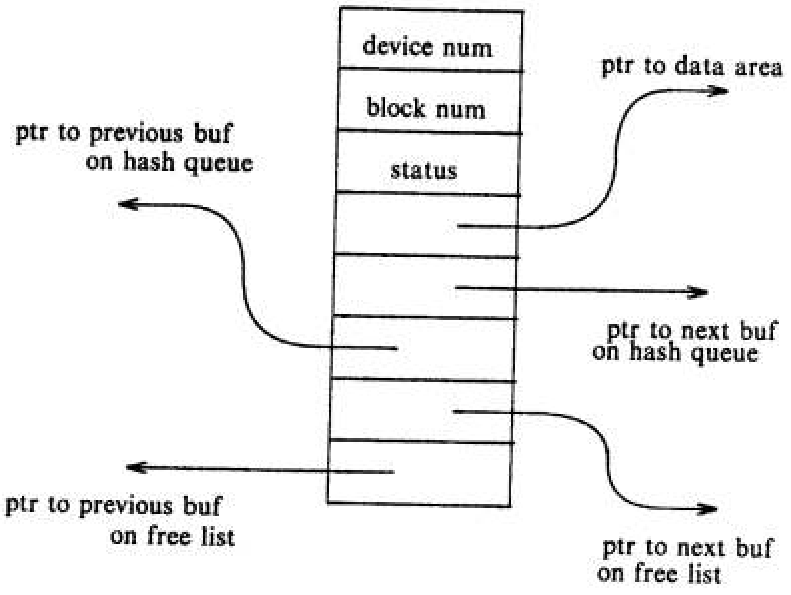
3.Define System Call.

System call are function invocations made from user space in order to request some services or resources from the operating system.

2.File and Directory I/O

1.Explain the structure of the Buffer Header.

When the system initializes the kernel allocates the space for the buffer cache. The buffer cache contains two regions/arts. One for the data/files that will be read from the disk, second the buffer header.


device num
block num
status
ptr to data area
ptr to previous buf on hash queue
ptr to next buf on hash queue
ptr to previous buf on free list
ptr to next buf on free list

The data in the buffer cache corresponds to the logical blocks of the disk block of file system. The buffer cache is "in memory" representation of the disk blocks. This mapping is temporary as the kernel may wish t load some other files' data into the cache at some later stage.

There will never be a case when the buffer has two entries for the same file on disk as this could lead to inconsistencies. There is only and only one copy of a file in the buffer.

The buffer header contains the metadata information like device number and the block number range for which this buffer holds the data. It stores the logical device number and not the physical device number. The buffer header also contains pointer to a data array for the buffer (i.e. pointer to the data region) .
The buffer header also contains the status of the buffer. The status of the buffer could be

- Locked/unlocked

- Buffer contains a valid data or not.
- Whether the kernel should write the contents to disk immediately or before reassigning the buffer(write delay)
- Kernel is currently reading the data or writing the data.
- Is there any process waiting for the buffer to get free.

3.Explain the structure of regular file with suitable diagram.
1.The inode consists of a table of contents to locates a file's data on disk where as the table of contents consists of a set of disk block numbers as each block on a disk is addressable by number.The inode is the data structure that describes the attributes of a file including the layout of its data on disk. There are two version of the inode: the disk copy that stares the inode information when the file is not in use and in-core copy that records information about active files.If the data in a file were stored in a contiguous section of the disk, then storing the start block address and the file size in the inode would suffice to access all the data in a file.

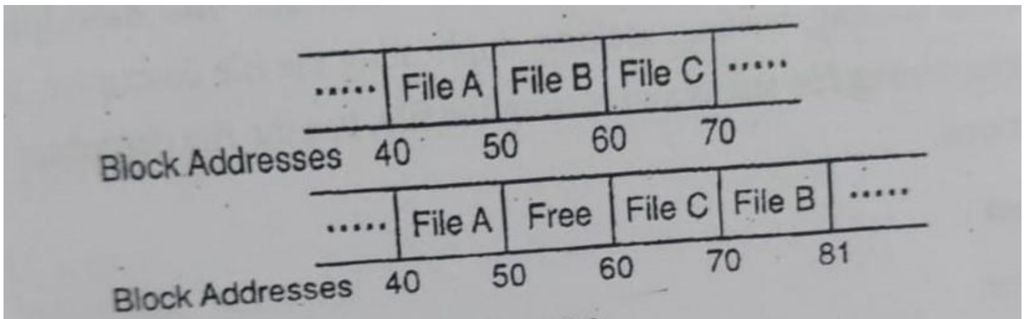


Figure 2.9

The above diagram shows the allocation of contiguous files and fragmentation of free space.The emol would have to allocate and reserve the contiguous space in the file system before allowing the operations that would increase the file size. The kernel could minimize fragmentation of the storage space by periodically running garbage collection procedures to compact available storage. The kamel allocates file space one block at a time and allows the data in a file to be spread throughout the file system 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 O 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 and continues to a logical block number corresponding to the file size.

4.Differentiate named and unnamed pipe.
Named pipes

1. They are created programatically using the command mkfifo.

2. They exists in the file system with a given file name.
3. They can be viewed and accessed by any two un-related processes. ls cmd shows “p” in the permission bits for a named pipe.
4. They are not opened while creation.
5. They are Bi-directinoal.

Un –named pipes

1. These are created by the shell automatically.
2. They exists in the kernel.
3. They can not be accesses by any process, including the process that creates it.
4. They are opened at the time of creation only.
5. They are unidirectional

5.Explain read(),open(),count(),write(),close() system call

1.open()

Open: The open system call specifies three parameters but programmers use only the first two. It is used to access the data in a file.

Syntax: fd open (pathname, flags, modes);

where pathname file name flags type (read/write)

and 'modes→ give the file permissions if the file being created.

The 'open' system call returns an integer called the user 'file descriptor' other file operations such as reading, writing, seeking, duplicating the file descriptor, setting I/O file parameters, determining file status and closing the file, use the file descriptor that the open system call returns.

2.read()

The syntax of the read system call is

Number = read (fd, buffer, count);

where, fd is the file descriptor returned by open, buffer is the address of a data structure in the user process that will contain the read data on the successful completion of the call.

3.count()

Count: Count is the number of bytes the user wants to read and the number is the number of bytes actually read.

algorithm read

input: user file descriptor

address of buffer in user process

number of bytes to read.

Output: Count of bytes copied into user space.

4.write()

The algorithm for writing a regular file is similar to that for reading a regular file.The syntax for the write system call,

Number = write(fd, buffer, count))

where, fd, buffer, count and number are the same as they are for the read system call.

5. close()

Close: A process closes an open file when it is no longer required. The syntax for the close system call is.close (fd); where, fd(file descriptor) is the file descriptor for the open file The kernel performs the 'close' operation by manipulating the file descriptors and the corresponding file tables and inode table entries.

6.Explain Reading and writing disk blocks.

To read block ahead, the kernel checks if the block is in the cache or not. If the block is not in the cache, it invokes the disk driver to read the block. Then the process goes to sleep awaiting the event that the I/O is complete. The disk controller interrupts the processor when the I/O is complete. The disk interrupt handler awakens the sleeping processes. The content of disk blocks are now in the buffer. When the process no longer needs the buffer, it releases the buffer so that other processes can access it Reading a Disk Block

```
 Input: file system block number

output: buffer containing data

{

get buffer for block (algorithm getbik):

if (buffer data valid) return buffer

initiate disk read; read

sleep (event disk complete);

 return (buffer);

}
```

To write a disk block the kernel informs the disk driver that it has a buffer whose contents should be output. The disk driver schedules the block for I/O. If the write is synchronous, the calling process goes the sleep awaiting I/O completion and releases the buffer when awakens. If the write is asynchronous, the kernel starts the disk write. The kernel releases the buffer when the I/O completes.

Writing a Disk Block

```
input: buffer

output: none

initiate disk write:

if (1/0 synchronous)

{
```

sleep (event 1/0 complete);

release buffer (algorithm brelse);

}

else if(buffer parked for delayed write)

mark buffer to put at head of free list;

}

7.Write any four operations related to inode.

1.iget -> Returns a previously identified inode.

2. input -> Releases an inode.

3.bmap-> Sets kernal parameters for accessing a file.

4. namei-> Converts a user level path name to an inode

8.Explain Mounting and unmounting file system.

The 'mount' and 'umount' system calls connect and disconnect a file system from the hierarchy.

Mounting file system

The 'mount' system call allows users to access data in a disk section instead of a sequence of disk blocks.

Syntax: mount (special pathname, directory pathname, options)] Here,

special pathname: name of the special device file of the disk section containing the file system to be mounted.

directory pathname: directory in the existing hierarchy where the file system will be mounted (called mount paint)

Option: indicate whether the file system should be mounted "read-only".

Unmounting File System

The system call unmount() is used to unmount file. The syntax for the unmount system call is. umount (special filename);

where, special filename indicates the file system to be unmounted. During un-mounting, the kernel access the inode of the device to be unmounted and retrieves the device number for the special file. It releases the inode and finds the mount table entry having device number equal to that of special file.

9.Explain Different type of file type.

1.Regular file: The most common type of file, which contains data of some form. There is no distinction to the UNIX kernel whether this data is text or binary. Any interpretation of the contents of a regular file is left to the application processing the file.

2. Directory file: A file that contains the names of other files and pointees to information on these files. Any process that has read permission for a directory file can read the contents of the dirctory, but only the kernel can write directly to a directory file. Processes must use the functions described in this chapter to make changes to a directory.

3.Block special file: A type of file providing buffered I/O access in fixed-size units to devices such as disk drives.

4. Character special file: A type of file providing unbuffered I/O access in variable-sized units to devices. All devices on a system are either block special files or character special files.

5.FIFO: A type of file used for communication between processes. It is sometimes called a named pipe.

6. Socket: A type of file used for network communication between processes. A socket can also be used for non-network communication between processes on a single host. We use sockets for interprocess communication.

7. Symbolic link: A type of file that points to another file.The type of a file is encoded in the st_mode member of Oct. 2017-2M What is symbolic link? the stat structure. We can determine the file type with the macros shown in the following table. The argument to each of these macros is the st_mode member from the stat structure.

10.unmask() function

The umask() system call sets the file mode creation mask for the process and returns te previous value. (This is one of the few functions that doesn't have an error return.)

Syntax: Mode_t unmask(mode_t cmask);

Most users of UNIX systems never deal with their umask value. It is usually set once, on login, by the shell's start-up file, and never changed. Nevertheless, when writing programs that create new files, if we want to ensure that specific access permission bits are enabled, we must modify the umask value while the process is running.For example, if we want to ensure that anyone can read a file, we should set the umask to 0. Otherwise, the umask value that is in effect when our process is running can cause permission bits to be turned off.

11.mmap() system call.

The kernel provides an interface which allows an application to map a file into memory. The can be done using the map) system call for mapping objects into memory and a file is accessed directly through memory.

#include <sys/anan.b>

void map (void addr, size_t len, int prot, int flags, int fd, offffset);

Here, fd→ file descriptor

offset→ starting offset bytes

fleag→ length bytes of the object

prot-access permissions

flags→ additional behaviour

Addr→ a preference to use that starting address in the memory.

12.Explain I/O schedulers.

The modern operating system kernels implement the I/O schedulers which minimize the and size of disk seeks by manipulating the order in which I/O requests are serviced and the times at which they are serviced.

1.Disk addressing: A hard disk consists of multiple platters, each consisting of a dick. spindle and read/write head. Each platter is divided into concentric circles like tracks and each track is then divided into a number of sectors. To locate a specific unit of data o disk, the drive's logic requires three information components such as cylinder, head and sector values.

The modern disk drives map a unique block number (also called physical blocks or device blocks) over each cylinder head sector. Then the disk is addressed using these block numbers. This process is called as LBA (Logical Block Addressing) and the hard drive internally translates the block number into the correct CHS (cylinder, head and sector) address.

2.The life of an I/O scheduler: Two basic operations are performed by an I/O scheduler namely: merging and sorting. Merging combines two or more adjacent DO requests into a single request, whereas sorting arranges the I/O operations in the increasing block order.Because of these two operations, the disk head's movements are minimized.

3.Optimizing I/O Performance: VO performance can be optimized by considering the ollowing factors such as:

a Using user buffering

b. Performing block-size aligned I/O

C. Using vectored I/O

d Asynchronous 1/0

13.Write a c program to print size, type of file.

#include<stdio.h> #include<io.h>

#include<fcntl.h>

#include<sys\stat.h> int main()

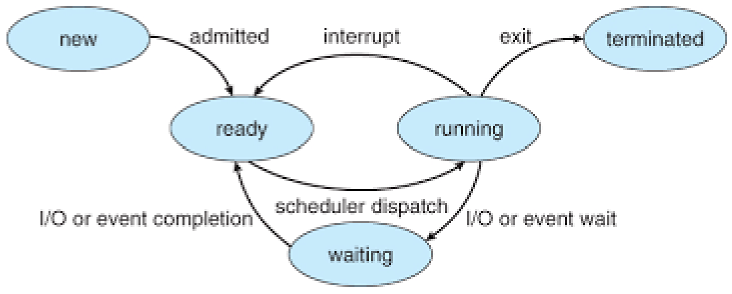
```
{
int fpi

long file size;
if((fp-open("f:/cprojects/urls.txt", O_RDONLY)) == -1)
printf("Error opening the file \n");
else
{
file size filelength (file_handle); printf("The file size in bytes is 1d\n", file size);
close (fp):
} return 0;
}/
```

3.Process Environment ,Process Control and Process Relationships.

1.Explain Process state.

The process state consists of everything to resume the process execution if it is somehow put aside temporarily.



A process goes through a series of discrete process states:

1.New state: Process being created.

2.Running state: A process is said to be running if it has the CPU, that is, process actually using the CPU at that particular instant.

3.Blocked (or waiting) state: A process is said to be blocked if it is waiting for some event to happen so that an I/O completion before it can proceed.

4.Ready state: A process is said to be ready if it uses a CPU if one were available. A ready state process is runnable but temporarily stops running to let another process run.

5.Terminal state: The process has finished execution.

2.Explain exit() and atexit() system call.

Exit Functions

A process can be terminated normally in five ways:

1.Executing a return from the main() function. This is equivalent to calling exit.

2. Calling the exit() function.

3.Executing a return from the start routine of the last thread in the process.

4. Calling the pthread-exit() function form the last thread in the process.

5.Calling the exit or exit() function

atexit-register a function to be called at normal process termination

Syntax is as follows:

```
#include <stdlib.h> int atexit (void (*function) (void)];
```

The atexit() function registers the given function to be called at normal process termination, either via exit or via return from the program's main(). Functions so registered are called in the Teverse order of their registration; no arguments are passed. The same function may be registered multiple times it is called once for each registration. The atexito function returns the value 0 if successful; otherwise it returns a non-zero value.

```
#include<stdio.h> Includesatdlib.h>

#include <unistd.h>

void bye (void) {printf("that was all, folks\n")}

int main(void)

{ Long a;

 int I;

a= sysconf (SC ATEXIT MAX);

printf("ATEXIT MAX-1d\n", a);

i= atexit (bye);

if(i!=0)

{fprintf(stderr, "cannot set exit function\n"); exit (EXIT FAILURE);

exit (EXIT FAILURE):

}exit (EXIT SUCCESS):

}
```

3.Explain wait() and waitpid() system call.

The wait() function suspends the execution of the calling process until one of its child terminates or was stopped by a signal or resumed by a signal.
In the case of fork() API, a child process is created which has its life cycle and runs independently.

If the parent process happens to terminate before the child process, the child process becomes orphan and gets re-parented by the Init process.
Syntax:
pid_t wait (int *status);

The waitpid() API suspends the execution of the caller process until a child specified by PID argument has either terminated or signaled to stop or resume.
The call wait() is equivalent to waitpid (-1, &status, 0).
By default, waitpid() waits only for the terminated child, not for the signaled child but this can be changed by the use of options.
The most important option is WNOHANG, which makes waitpid() a non-blocking call.
Syntax:
int waitpid(int pid, int * stat_var, int options);

4.state and explain setjump() and longjump() functions.
setjmp and longimp0) are subroutines that perform complex flow-of-control in C/Unix These functions are used for exception handling. They basically allows the user to discard bunch of stack frames and try to create fault-tolerance boundaries, but not in every function.
One of the keys to understand setjmp0) and longimp0) is to understand machine layout, as described in the assembler. The state of a program depends completely on the contents of its memory (i.e. the code, globals, heap and stack) and the contents of its registers. The contents of the registers includes:
The Stack Pointer (SP)
Frame Pointer (FP)
Program Counter (PC) Process State Register (PS)
What setimp does is save the contents of the registers so that longimpo) can restore them later. In this way, longimpo "returns" to the state of the program when setjmp was called.

Specifically:

#include<set jmp.h> int set jmp (jmp_buf env);

longjmp (jmp_buf env, int vall);

5.Explain Daemons?

A daemon is a process that runs in the background, not connecting to any controlling terminal. Daemons are normally started at a boot time, run as root or some other special user and handle system level tasks.
A daemon has two general requirements such as:
1. It must run as a child of init.
2.It must not be connected to a terminal.
In general, a program, performs the following steps to become daemon:
1.Call fork (: This creates a new process which will become the daemon.
2.Call exit(): This ensures that the original parent is satisfied and that its child terminated The daemons parent is no longer running and that the daemon is not a process group leader.
3.Call setsid: Giving the daemon, a new process group and session, both of which have in as leader.
4. Change the working directory to the root directory via chdir(). This is done because the inherited working directory can be anywhere on the file system. Close all file descriptors.
5 Open file descriptors 0, 1 and 2 (standard in, standard out and standard error) and redirect them to devioull.

6.Explain Processor affinity.
Processor affinity refers to the likelihood of a process to be scheduled consistently on the same processor. The Linux scheduler attempts to schedule the same processes on the same processor for as long as possible, migrating a process from one CPU to another only in situation of extreme load imbalance. The term soft affinity refers to the above mentioned situation.
Sometimes, however, the user an application wants to enforce a process to-processor bond Bonding a process to a particular processor and having the Kernel enforce the relationship is called setting a hard affinity.A process has affinity for processor on which it is currently running.

7.Explain Orphan and zombie process.
It is a computer process whose parent process and process terminated, though itself remains running.
In UNIX/LINUX operating system, any orphaned process will be immediately adopted by the special init system process. This operation is called as re-parenting and occurs automatically. A process can be orphaned unintentionally such as when the parent process terminates or crashes. The process group mechanism can be used to protect against accidental orphaning. where in coordination with the user's shell will try to terminate all the child processes with the SIGHUP process signal rather than letting them run as orphans. A

process may also be intentionally orphaned making the process run in the background. With remote invocation a server process is also said to be orphaned. When the client that initiated the requests unexpectedly crashes after making the request while leaving the server process running.
These orphaned processes waste server resources and can potentially leave a server starved for resources. The orphan process problem can be resolved by: By killing the orphan (extermination) Expiration: each process is allotted a certain amount of time to finish before being killed. Machines try to locate the parents of any remote computations at which point orphaned processes are killed (Reincarnation). A process that has terminated, but that has not yet been waited upon by its parent is called a "zombie." Zombie processes continue to consume system resources, although only a small percentage enough to maintain a mere skeleton of what they once were.

4.Memory Management

1.Explain Anonymous Memory Mapping

The algorithm is known as "buddy memory allocation scheme' which works using two types of fragmentation:

1. Internal fragmentation: It occurs when more memory than requested is used to satisfy an allocation. This result in inefficient use of the available memory.

2.External fragmentation: It occurs when sufficient memory is free to satisfy a request but it is split into two or more nonadjacent chunks. This can result into inefficient use of memory (because a larger, less suitable block may be used) or failed memory allocation. Allocating memory via anonymous mappings has several benefits:

a. No fragmentation concerns: When the program no longer needs an anonymous memory mapping, the mapping is unmapped and the memory is immediately returned to the system.

b. Anonymous memory mappings are resizable and have adjustable permissions and can receive advice just like pok

C Each allocation exists in a separate memory mappings. There is no need to manage the global heap.

2.Explain Data Segment

A data segment is a portion of virtual address space of a program which contains the global variables and static variables that are initialized by the programmer. This potion has a fived si for each programs depending upon the quanuity of comments, since all of the data here is vet by the programmer before the program is loaded.

Data

The data area contains global and static variables used by the program that are initialized. This segment can be further classified into initialized read-only area and initialized road-write area.

BSS: The BSS segment also collectively known as uninitialized dan starts at the end of the data segment and contains all global variables and static variables that are initializel to zero or do not have explicit initialization in source code. For instance a variable declared static int 1 would be contained in the B33 segment

Heap: The heap area begins at the end of the BSS segment and grows to larger addresses from there. The Heap area is managed by mallos, reallos, and free, which may use the brk and abrk() system calls to adjust its size (Note: The use of bekchbek and a single "heap area" is not required to fulfill the contract of malloo/realloo free; they may also be using mmap to reserve potentially non-contiguous regions of virtual memory into the process' virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

Stack

The stack area traditionally adjoined the heap area and grew the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and vinual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions)

3.Explain Anonymous Memory Mappings.

The algorithm is known as 'buddy memory allocation scheme' which works using two types of fragmentation:

Internal fragmentation: It occurs when more memory than requested is used to satisfy an allocation. This result in inefficient use of the available memory.

External fragmentation: It occurs when sufficient memory is free to satisfy a request but it is split into two or more nonadjacent chunks. This can result into inefficient use of memory (because a larger, less suitable block may be used) or failed memory allocation. Allocating memory via anonymous mappings has several benefits:

a. No fragmentation concerns: When the program no longer needs an anonymous memory mapping, the mapping is unmapped and the memory is immediately returned to the system.

b. Anonymous memory mappings are resizable and have adjustable permissions and can receive advice just like son.

4.State and explain the param parameter value.

The advanced memory allocation can be done by mallopt system call

#include <malloc.h>

int mallopt (int param, int value);

Linux supports the following six values for param' defined in smalloco:

1.M CHECK ACTION MMAP MAX

2.M MAP THRESHOLD

3.MMXFAST

4.M TOP PAD

5. M TRIM THRESHOLD

1.M_CHECK_ACTION: Setting this parameter controls how glibe responds when various kinds of programming errors are detected.

2.M MMAP MAX: The maximum number of mappings that the system will make to satisfy dynamic memory requests. When this limit is reached, the data segment will be used for all allocations, until one of these mappings is freed. A value of 0 disables all use of anonymous mappings as a basis for dynamic memory allocations.

3. M MMAP THRESHOLD: The size threshold (measured in bytes) over which an allocation request will be satisfied via an anonymous mapping instead of the data segment. Note that allocations smaller than this threshold may also be satisfied via anonymous

4.M MXFAST: The maximum size (in bytes) of a fast bin. Fast bins are special chunks of memory in the heap that are never coalesced with adjacent chunks, and never returned to the system, allowing for very quick allocations at the cost of increased fragmentation. A value of 0 disables all use of fast bins.

5.M TOP PAD: The amount of padding (in bytes) used when adjusting the size of the data segment. Whenever glibc uses bek() to increase the size of the data segment, it can ask for more memory than needed, in the hopes of alleviating the need for an additional brk() call in the near future.

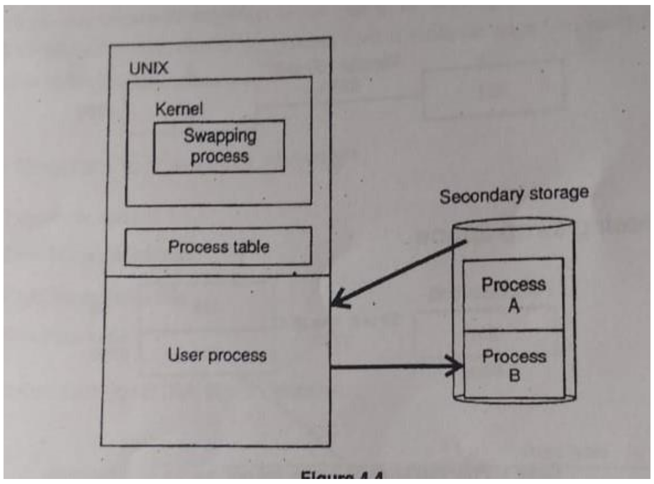
6. M TRIM THRESHOLD: The minimum amount of free memory (in bytes) allowed at the top of the data segment. If the amount falls below this threshold, glibc invokes brk() to give back memory to the kernel.

5.Explain swapping Process.

1.When a process first enters real memory, the entire image is loaded. As the process grows, new primary memory is allocated, the process is copied to the new space and the process table is updated. If sufficient memory is not available, the growing process is allocated space on secondary memory and swapped out. At this point, the process can reside on secondary memory.

2. The swapping process is part of the kernel in above diagram it is shown, so it can be activated each time UNIX gets control. It scans the process table, looking for a ready process that has been twapped out. If it finds one, it allocates primary memory and swaps in the process. If insufficient memory space is available, the swapping routine selects a process to be swapped out, copies the selected process to secondary storage, frees the memory space, and then swaps in the ready process.

3.The swap-in decision is based on secondary storage residency time-the longer a process resides on disk, the higher its priority. Generally, processes waiting for slow events are primary swap out candidates; if there are several such processes, age in primary memory is a secondary criterion. A slight penalty is imposed on large programs. To help minimize thrashing, processes do not become candidates for swapping out until they have achieved at least a minimum age in primary memory. Early versions of UNIX uses swapped segments, newer versions designed to run on page-oriented hardware subdivide segments into pages and swap pages.



1. The swap device is a block device in a configurable section of a disk.

2.Kernel allocates contiguous space on the swap device without fragmentation.

3.It maintains free space of the swap device in an in-core table, called map.

4. The kernel treats each unit of the swap map as group of disk blocks.

5.As kernel allocates and frees resources, it updates the map accordingly.

6.State and explain the data structure used for demand paging.

Virtual memory uses a technique called paging', the virtual address space is divided into units called pages. The corresponding units in physical memory MMU is responsible for mapping virtual addresses into physical addresses by means of the TLB.

Demand Paging is the process of bringing pages into memory according to need and allows system to treat RAM as a cache for the disk. It relies on locality of reference, temporal locality one page used multiple times within a period of time.

kernel suspends the execution of the process until it reads the page into memory and makes it When a process accesses a page that is not part of its working set, it incurs a page fault. The accessible to the process.

Data Structure for Demand Paging

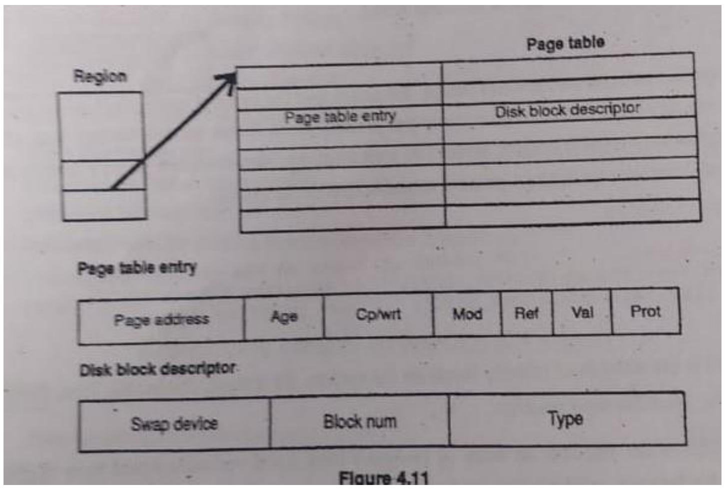
1.Page table entry

2. Disk block descriptors

3.Page frame data table

4. Swap use table

Page Table Entry and Disk Block Descriptor



Page Table Entry

It contains the physical address of page and the following bits:

1.Valid: whether the page content is legal.

2.Reference: whether the page is referenced recently. iii. Modify: whether the page content is modified.

3.Copy on write: kemel must create a new copy when a process modifies its content (required for fork).

4.Age: Age of the page.

5.Protection: Read/ write permission.

5.signal Handling

1.Explain kill() and rise() system call.

A process may send a signal to itself by calling raise(). A process may send a to another process, including itself, by calling kill.

Int kill(int pid, int signal)

A system call that send a signal to a process, pid. If pid is greater than zero, the signal is sent to the process whose process ID is equal to pid. If pid is 0, the signal is sent to all processes, except system processes.

kill() returns 0 for a successful call, -1 otherwise and sets errno accordingly.

int raise(int sig)

It sends the signal sig to the executing program. raise () actually uses kill () to send the signal to the executing program:

kill (getpid(),sig); There is also a UNIX command called kill that can be used to send signals from the command line.

2.Explain sigqueue() function.

Signal handlers registered with the SA SIGINFO flag are passed a siginfo_t parameter. This structure contains a field named si-value, which is an optional payload passed form the signal generator to the signal receiver.

A function known as sigqueue((defined by POSIX) allows a process to send a signal with this payload.

#include <signal.h>

int sigqueue (pid t pid, int signo, const union signal value);

Here, the function sigqueue() works similar to kill. The signal's payload is given by value. which is an union of an integer value and a void pointer described as follows:

union sigval

{int sival_int;

void sival_ptr:}

Example

Using payload signals

```
saigval val;

int t;

 val.aival_int-404;

t=sigqueue (1722, SIGUSK2, val);

if(t) perror("aigqueue"))
```

The above example, sends the process with pid (1722) the SIGUSR2 signal with a payload of an integer that has a value 404.

3.Explain in advanced signal management.

The signal() function is very basic. Because it is part of the standard C library, and therefore has to reflect minimal assumptions about the capabilities of the operating system on which it runs, it can offer only a lowest common denominator to signal management.

POSIX standardizes the sigaction() system call, which provides mich greater signal management capabilities. Among other things, it is used to block the reception of specified signals while your handler runs, and to retrieve a wide range of data about the system and process state at the moment a signal was raised:

```
#include <aignal.h Int sigaction (int signo, const struct sigaction act, struct sigaction oldact);
```

A call to sigaction() changes the behavior of the signal identified by signo, which can be any value except those associated with SIOKILL and BIOSTOP. If act is not NULL, the system call changes the current behavior of the signal as specified by act. If oldact is not NULL, the call stores the previous (or current, if act is NULL) behavior of the given signal there.

The function receives the signal number as its first parameter, a siginfo_t structure as its second parameter, and a ucontext I structure (cast to a void pointer) as its third parameter. It has no return value. The siginfo_t structure provides an abundance of information to the signal handler.

4.Explain sigsuspend(), abort() , system() and sleep() function.

1.sigsuspend() function: Sigsuspend resets the signal mask as well as puts the process to sleep in a single atomic operation,

#include <signal.h int algsuspend (const algsett algmaak))

The signal mask of the process is set to the value pointed to by sigmask. Then the proces is suspended until a signal is caught or until a signal occurs that terminates the process. If a single is caught and if the signal handler returns, then sigsuspend returns and the signal mask of the process is set to its value before the call to sigsuspend.

2.abort() function: It results in abnormal program termination. This function sends the SIGABRT signal to the caller processes. When the process catches the SIGABRT signal, it allows the SIGABRT to perform the clean up operation before the process terminates.

#include<stdlib.h void abort (void);

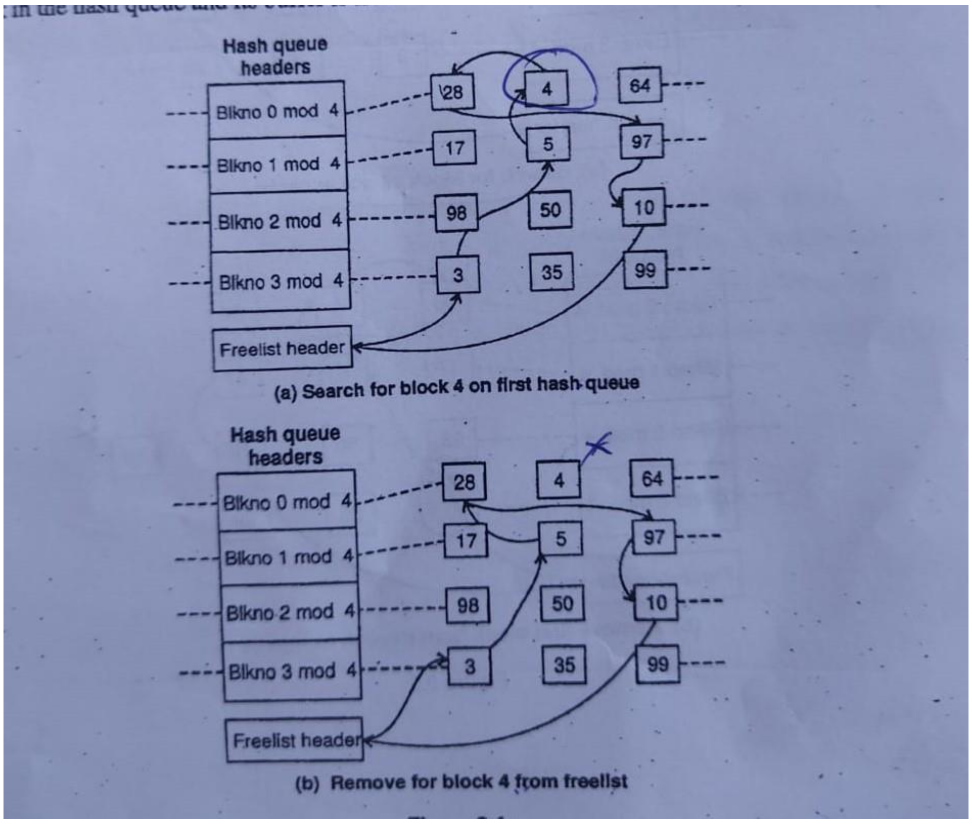
This function never returns.

3.system function: The purposes to handle signals. When it is required to include a system function in a program, it is required to interpret the correct return value as it is the termination status of the shell which is not always the termination, status of the command string

4. sleep() function: Sleep function causes the calling process to be suspended until either the amount of wall clock time specified by seconds has elapsed or a signal is caught by the process and the signal handler returns.

***scenarios of buffer.**

1.Block in the hash queue and its buffer is free.



2.cannot find block on the hash queue=>allocate a buffer from free list.

3.cannot find block on the hash queue=> allocate a buffer free list but buffer on the free list marked ‘delayed write’=>flush delayed write buffer and allocate another buffer.

4.cannot find block on the hash queue and free list of buffer also empty.

5.Block in the hash queue but buffer is busy.