AMMINI COLLEGE OF ENGINEERING

CS09 607 (P)

SYSTEMS LAB – OS (A)

LAB MANUAL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CYCLE I

CYCLE II

OPERATING SYSTEM CONCEPTS

An operating system is the interface between hardware and the users. An operating system provides the environment within which programs are executed. Internally operating system varies generally in their makeup, being organized along many different lines.

SYSTEM CALLS

System calls is a technique to a program executing in user mode can request the kernel's services. For the system call approach the user process uses the process trap instruction. The ideas that the system call should appear to be an ordinary procedure call to the application program. The OS provides a library of user functions with names corresponding to each actual system call. Each of these "stub functions" contains a trap to the OS function. When the application program calls the stub, it executes the trap instruction, which switches, the CPU to supervisor mode, then branches indirectly through an OS table to the entry point of function that is to be invoked. When the function completes, it switch the processor to the user mode and these returns control to the user process. File system call and relation to other algorithms are classified the system calls in to several categories, although some system calls appear in more than one category.

System calls that return file descriptor for use in other system calls.

System calls that use the naming algorithm to parse a path name.

System calls that assign and free in order, using algorithms ialloc and ifree.

System calls that set or change the attributes of a file.

System calls that do i/o to and from a process using algorithms alloc, free and other buffering
Algorithms.

System calls that change the structure of a file system.

System calls that allow a process to change its view of the file system tree.

The system calls are OPEN, READ, WRITE, LSEEK and CLOSE.

INTERPROCESS COMMUNICATION

Inter process communication mechanism allow attribute processes to exchange data and synchronize execution. Pipes message, shared memory are some form of interprocess communication.

PIPES

Pipes allow transfer of data between processes in first-in-first-out (FIFO) manner and they also allow synchronization of process execution. Their implementation allows process to communicate even though they do not know that what processes are on the other end of the pipe. The traditional implementation of pipes uses the file system for data storage. There are two kinds of pipes named pipes and for lack of a better term, unnamed pipes, which are identical except for the way that a process initially accesses them. Process uses the open system call for named pipe system call to create an unnamed pipe. Afterwards, processes use the regular system calls for files such as read, write and close when manipulating pipes. Only related processes, descendants of a process that issued the pipe system call, can share access to unnamed pipe.

SHARED MEMORY

Process can communicate directly with each other by sharing parts of their virtual address space and then reading and writing the data shared in the shared memory. The system calls for manipulating shared memory are similar to the system calls for messages. The shmget system call creates a new region of system shared memory or returns an existing one , shmat system call logically attaches a region to the virtual address space of a process, the shmdt system call detaches a region from the virtual address space of a process, and the chmct1 system call manipulates varius parameters associated with the shared memory. Processes read and write shared memory using the same machine instructions they use to read and write regular memory. After attaching shared memory, it becomes part of the virtual address space of a process, accessible in the same way other virtual addresses are; no system calls are needed to access data in shared memory.

The syntax of the shmget system call is

Shmid = shmget (key,size,flag);

where size is the number of bytes in the region. The kernel searches the shared memory table for the given key: if it finds an entry and the permission modes are acceptable, it returns the descriptor for the entry. If it does not finds an entry and the user had set the the IPC_CREATE flag to create a new region, the kernel verifies that the size is between system-wide minimum and maximum values an d then allocates a region data structure using allocreg.

A process attaches a shared memory region to its virtual address space with the shmat system call:

Virtaddr = shmat (id,addr,flags);

Id, returned by a previous shmget system call, identifies the shared memory region, addr is the virtual address space where the user wants to attach the shared memory, and flags specify whether the region is read-only and whether the kernel should round off the user specified address.

A process detaches a shared memory region from its virtual address space by Shmdt(addr)

Where addr is the virtual address returned by a prior shmat call.

A process uses the shmct1 system call to query status and set parameters for the shared memory region:

Shmctl(id, cmd,shmstat(buf);

Id specifies the shared memory table entry, cmd specifies the type of operation. And shmstatbuf is the address of a user-level data structure that contains the status information of the shared memory table entry when querying or setting its status.

MESSAGE PASSING

There are four system calls for messages. Msgget returns (and possibly create) a message descriptor that designates a message queue for use in other system calls, msgct1 has options to set and return parameters associated with a message descriptor and an option to remove descriptors, msgsnd sends a message, and msgrcv receives a message.

The syntax of the msgget system call is

msgqid = msgget(key,flag); (Where msgqid is the descriptor returned by the call.)

A process uses the msgsnd system call to send a message

Msgsnd (msgqid, msg, count, flag);

Where msgqid is the descriptor of a message queue typically returned by a msgget call, msg is a pointer to a structure consisting of a user- chosen integer type and a character array, count gives the size of the data entry and flag specifies the action the kernel should take if it runs out of internal buffer space.

A process receives messages by

Count = msgrcv(id, msg, maxcount, type, flag);

Where id is the message descriptor, msg is the address of a user structure to contain the received message, maxcount is the size of the data array in msg, type specifies the message type the user wants to read, and flag specifies what the kernel should do if no messages are 0on the queue. The return value, count, is the number of bytes returned to the user.

A process can query the status of a message descriptor, set its staus, and remove a message descriptor with msgct1 system call. The syntax of the call is:

Msgctl(id,cmd,mstatbuf)

Where id identifies the message descriptor, cmd specifies the type of command, and mstatbuf is the address of a user data structure that will contain control parameters or the results of a query.

PROCESS SYNCHRONIZATION

When two or more threads need to access to a shared resource, they need some way to ensure that the resource will be used only one thread at a time. The process by which this is achieved is called synchronization. Semaphores allow process to synchronize execution.

SEMAPHORES

The semaphore system calls allow processes to synchronize execution by doing a set of operations atomically on a set of semaphores. The implementation of Semaphores defines two integer – valued objects that have two atomic operations define for them. The P operation decrements value of a semaphore if its value greater than 0, and V operation increments its value. Because the operations are atomic, at most one P or V operation can succeed on a semaphore at any time.

The semaphore system calls are scmget to create and gain access to a set of semaphores, scmctl to do various control operations on the set, and scmop to manipulate the value of semaphores.

DEADLOCK

Deadlock is a situation in which two or more process get in to a state where by each is holding a resource, the other is requesting. Since the request operation blocks the caller until the resource become allocated, neither process ever have desired resources allocated to it and both will remain in the blocked state for ever.

There are four approaches by which the operating system handles deadlock.

a) Prevention

There are four necessary but not sufficient conditions for deadlock to exist.

Mutual exclusion

Once a process has been allocated a particular resource, it has exclusive use of the resource. No other process can use a resource while it is allocated to a process.

Hold and wait

A process may hold a resource at the same time it request another one.

Circular waiting

A situation can arise in which p holds resource r1 while it request resource r2, and a process p2 holds r2, while it request resource r1. There may be more than two process involved in circular wait.

No preemption

Resources can be released only by the explicit action of the process rather than by the action of an external authority.

Deadlock can be prevented by violating these conditions.

b) Avoidance

Avoidance strategies rely on manager's ability to predict the effect of satisfying individual allocation request. If a request can lead to a situation in which a deadlock could occur, avoidance strategies will refuse the request. Since avoidance is a predictive approach, it relies on information about the resource activity that will be occurring for the process. Avoidance is a conservative strategy, it tends to under utilize resources by refusing to allocate them if there is a potential for a deadlock.

c) Detection and recovery

In this approach the system checks to see if deadlock exist, either periodically or whenever certain events occur. There are two phases for detection strategy. The first is the detection phase, during which the system is checked to see if a deadlock currently exist. If a deadlock is detected, the system goes through second phase, by preempting resources from processes. The detection and recovery strategy is most widely used deadlock strategy.

d) Manual deadlock management

When a deadlock occurs in a system, it is up to the user or operator to detect it.

PROCESS SCHEDULING

The scheduling mechanism handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

On a time sharing system, the kernel allocates the CPU to a process for a period of time slice or time quantum, preempts the process and schedules another one. When the time slice expires and reschedules the process to continue execution at later time. The scheduler time function on the UNIX system uses relative time of schedules next. Every active process has a scheduling priority; the kernel switches context to that of the process the highest priority, when it does a context switch. The kernel recalculates the priority of a running process, when it returns from kernel mode.

IMPLEMENTATION OF CP COMMAND

AIM :

To write a program to copy an existing file to a new file using the system call open, read and write.

DESCRIPTION

The program makes a new copy of an existing file using the system call open, create and read.

int open (char filename, int mode)

open : Opens the specified file with the specified mode. The different modes are O_RDONLY, O_WRONLY,O_RDWR. Open returns a file descriptor for use in other system calls.

int creat (char filename, int mode)

creat : Creates a new file with the indicated file name and access permission modes. If the file already exists, create truncates the file. Create retrieves a file descriptor for used in other system calls.

int read (int fields , char buf , int size)

read: reads up to size bytes from the file fields into the user buffer buf. Read returns the number of bytes it read.

exit (int status)

exit: Causes the calling process is terminate reporting the low order bits of status into waiting parent.

ALGORITHM:

```
Step 1:  Start
Step 2 :  Read the command line arguments
Step 3 : If the number of arguments is not equal to 3 then print an error
            Message.
Step 4 : Open the source file
Step 5 : Create the destination file
Step  6: Call the copy function
Step  7: In the call function, data is read from the source file and is written to the
            destination file.
Step 8: Stop
```

CREATION OF A NEW PROCESS TO COPY FILES

AIM :

To write a program to create a new process to copy files by using fork, execl, wait system calls.

DESCRIPTION

fork ()

fork creates a new process. The child process is a logical copy of the parent process, except that the parent's return value from the fork is the process ID of the child and the child's return value is 0.

wait (int * wait-stat)

wait causes the process to sleep until it discovers a child process that has exited or a process sleep in tree mode. If wait state is not 0, it points to an address that contains status information on return from the call.

execl (filename, argv, exvp)

execl executes the program file filename, overlaying the address space of the executing process. Argv is the array of characters string parameters; exvp is the environment of new process.

ALGORITHM

```
Step 1: Start
Step 2: Create a new process using fork system call.
Step 3: The new process invokes execl is execute the program copy.
Step 4: Wait system call is called.
Step 5: Stop.
```

LS COMMAND IMPLEMENTATION

AIM

To write a program to implement LS command.

DESCRIPTION

The **dirent** structure is declared as follows:

```
struct dirent
{
        long   d_ino;          //inode number

        off d_off ;            //offset to this dirent

        unsigned short d_reclen ;          // length of this d_name

        Char d_name [NAME_MAX + 1];        // file name [null.terminated]
 }
```

DIR * opendir (const char *name)

The opendir () function opens a directory stream corresponding to the directory name and
Returns a pointer to the directory stream or null if an error occurred .

Struct dirent * readdir (DIR * dir)

The readdir function retrieves a pointer to a dirent structure representing the next directory entry in the entry stream pointed to by dir. It returns on reaching the end of the file or of an error occurred.

ALGORITHM

```
Step 1 :  Start.
Step 2 :  Read the directory name as command line arguments.
Step 3 :  Open the specified directory , which returns a value to the pointer dp
              to DIR.
Step 4 :  If dp = null , goto step 9 , else goto 5.
Step 5 :  Read the contents of the directory pointed to by dp till the pointer to
              the struct dirent returns NULL.
Step 6 : Print the contents of the directory.
Step 7 : Close the directory.
Step 8 : Go to step 10.
Step 9 : Print “cannot open the directory”.
Step 10 : Stop.
```

NAMED PIPE

AIM

To create a named Pipe with read – write permission for all users.

DESCRIPTION:

The system calls used in the program are

mknod (filename, modes, dev)

char * filename;

int mode, dev ;

mknod creates a special file , directory or FIFO according to the specified type of mode. If the file in block special or character special, dev gives the major and minor numbers of the device.

open (char * filename , int mode)

open opens the specified file according to the value of the mode.

read (int fields , char * buf, int size)

read reads up to size bytes from the file the user buffer buf. It returns the number of bytes it need.

write (int fd , char * buf , int count)

Write writes count bytes of data from user address buf is the file whose descriptor is fd

ALGORITHM:

```
Step 1 : Start.
Step 2 :Read the command line argument .
Step 3 :Create a pipe named FIFO using the system call mknod( ).
Step 4 : If the number of arguments=2 , open the pipe in write only mode.
Step 5 : Else open the pipe in read only mode.
Step 6 : if the argument count = 2, write the string into a buffer and print it.
Step 7 : Else read the string from the buffer.
Step 8 : Repeat steps 6 and 7 four times.
Step 9 : Stop.
```

MEMORY SHARING

AIM

To write a program to implement shared memory mechanism to attach data into shared memory area and create another process to share the data.

DESCRIPTION

Process can communicate directly with each other by sharing parts of the virtual address space and then reading and writing data stored in the shared memory. There are four system calls for manipulating shared memory.The shmget system call create a more region of shared memory or returns an existing one ,the shmat system call logically attaches a region to the virtual address space of a process, the shmdt system call detaches a region from the virtual address space of the region of a process and the shmct1 system call manipulate various parameters associated with the shared memory.

The syntax of the system calls :

shmid=shmget(key,size,flag)

Key_tkey ;

int size, flag;

where the size is the number of bytes in the region .The kernel searches the shared memory table for the given key. If it finds, it returns a descriptor, flag can be either IPC_CREATE or IPC_EXCL.

virtaddr = shmat(id,addr,flag)

where id is the descriptor returned by shmget system call, addr is the virtual address where the user wants to attach the shared memory, and flag specify whether the region is read-only and whether the kernel should round off the user-specified address. The return value is the virtual address where the kernel attached the region.

shmdt(addr)

addr is the address returned by shmat system call.

shmctl(id,cmd,shmstatbuf);id specifies the shared memory table entry, cmd specifies type of operation and shmstatbuf is the address of the user level data structure that contains the status information.

ALGORITHM: SERVER PROGRAM

Step1: Start.

Step 2: For I=0 to 19, signal system call is called with cleanup() function.

Step 3: Create a 128 k byte shared memory region.

Step 4: Attach it twice to address space at different virtual addresses, addr1, addr2.

Step 5: Write data, to the first shared memory and read it from the second shared memory.

Step 6: Print the data.

Step 7: Pause system call is called to give the second process a chance to execute.

Step 8: Stop.

ALGORITHM:CLIENT PROGRAM

Step 1: Start

Step 2: Retrieve the shared memory of the first process by using shmget system call with the same
key.

Step 3: Attach the shared memory to its address space.

Step 4: Read data from the memory and print it.

Step 5: Stop.

MESSAGE PASSING

AIM

Using the system V IPC package, create a message queue that can be shared by another process.

DESCRIPTION

Messages allow procedures to send formatted data streams to arbitrary process. There are four system calls for messages.

Msgqid = msgget (key, int flag)

Msgget returns an identifier to a message queue whose name is key. Key can specify that the returned queue identifier should refer to a printer queues (IPC - PRIVATE), in which are a new message queue is created . Flag can be either IPC-CREATE or IPC- EXCL.

msgsnd (ink id, struct msgbuf * msgp, int size, int flag)

msgsnd sends a message of size bytes in the buffer msgp to the message queue id. Flag is used to set or reset IPC-NOWAIT.

Msgrcv(int id, struct msgbuf * msgp, int size, int type, int flag)

Msgrcv receives message from the queue identifier by id. If type is 0, the first message on the queue is received, if positive , the first message of that type is received , if negative the first message of lowest type less than or equal to type is received. Flag is used to set msg NO ERROR , IPC-NOWAIT.

Msgctl (int id, int cmd, struct msgid – ds * buf)

Msgctl allows process to set or query the status of the message queue id, or to remove the queue, according to the value of cmd.

ALGORITHM

```
Step 1 : Start
Step 2:  A Structure queue-structure is defined.
Step 3 :A function err-sys which prints out error-message is defined.
Step 4 : Define a function make queue which creates a queue using msgget system call.
Step 5 : Define a function name-to-num which converts a string into ASCII code.
Step 6 : Define a function send which sends a message using the system call msgsnd.
```

Step 7 : The receive function receive the message using msgrcv system call.

Step 8 : In the main function , the message is sent or received according to user's choice.

Step 9 : Stop.

DINING PHILOSOPHER'S PROBLEM

AIM

Write a program to implement dining philosopher's problem.

DESCRIPTION

The system calls used are

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void*), void *arg);
```

The *pthread_create()* function is used to create a new thread, with attributes specified by *attr*, within a process. If *attr* is NULL, the default attributes are used. If the attributes specified by *attr* are modified later, the thread's attributes are not affected. Upon successful completion, *pthread_create()* stores the ID of the created thread in the location referenced by *thread*.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

The *pthread_join()* function suspends execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated. On return from a successful *pthread_join()* call with a non-NULL *value_ptr* argument, the value passed to pthread_exit() by the terminating thread is made available in the location referenced by *value_ptr*. When a *pthread_join()* returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to *pthread_join()* specifying the same target thread are undefined. If the thread calling *pthread_join()* is canceled, then the target thread will not be detached.

sem_wait(sem_t *sem);

The semaphore referenced by sem is locked. When calling **sem_wait**(), if the semaphore's value is zero, the calling thread will block until the lock is aquired or until the call is interrupted by a signal. Alternatively, the **sem_trywait**() function will fail if the semaphore is already locked, rather than blocking on the semaphore. If successful (the lock was aquired), **sem_wait**() and **sem_trywait**() will return 0. Otherwise, -1 is returned and errno is set, and the state of the semaphore is unchanged.

int **sem_post**(sem_t *sem);

The the semaphore referenced by sem is unlocked, the value of the semaphore is incremented, and all threads which are waiting on the semaphore are awakened. **sem_post**() is reentrant with respect to signals and may be called from within a signal hanlder. If successful, **sem_post**() will return 0. Otherwise, -1 is returned and errno is set.

ALGORITHM

```
Step1 : Start.
Step 2: Create an array of semaphores for each philosophers and initialize them
Step 3: Create a process for suspending the program. The program get
        killed if Q is read from keyboard.
Step 4: For each philosophers, new process is created and the function philosopher is
        called.
Step 5: The function philosopher
        a)If it is not the last philosopher, then do the steps from b to f
        b)Pick up the left fork
        c)Pick up the right fork
        d)Eat
        e)Release the right fork.
        f)Release the left fork.
        g)In the case of last philosopher, the order of fork is removed.
Step 6: Stop.
```

PROCESS SCHEDULING

AIM:

To write a program to implement process scheduling algorithm.

DESCRIPTION:

CPU scheduling deals with the problem of deciding which of the process in the ready queue is to be allocated to the CPU.

First come first serve scheduling: The simplest CPU scheduling algorithm is the FCFS algorithm. With this scheme the process that requests the CPU first is allocated to the CPU first.

Shortest job first scheduling: This algorithm associate with each process the length of the letters next CPU burst. When the CPU is available it is assigned to the process that has the smallest next CPU burst. FCFS is used to break the tie.

Priority scheduling: The SJF is a special case of the general priority scheduling algorithm. A priority is associated with each process and the CPU is allocated to the process with the highest priority equal priority processes are scheduled in FCFS order.

Round robin scheduling: The round robin scheduling algorithm is assigned specially foe time sharing system. It is similar to FCFS scheduling but preemption is added to switch between processes. New processes are added to the ready queue. set a timer to the interrupt after time quantum and dispatches the process.

ALGORITHM:

Step 1 : Start

Step 2 : Read the number of processes.

Step 3 : Read the service time of each process.

Step 4 : In FCFS , the processes are scheduled in the order in which they arrive.

Step 5 : In shortest Job Next (SJN) , the process requiring minimum service time is chosen as its next one. Waiting time and turn around time of each process is Printed .Average waiting time and average turn around time is also found.

Step 6 : In priority scheduling , the processes according to their priority entered are scheduled. Waiting time and turn around time of each process is printed.

Average waiting time and turn around time is also found.

Step 7 : In round robin , each process is scheduled for a fixed time quantum, then it

removed from the processor and next process is scheduled. In this also average waiting time and turnaround time is found and printed.

Step 8 : Stop

DISK SCHEDULING

AIM:

To write a program to implement disk scheduling algorithms.

DESCRIPTION:

Disk scheduling is one of the responsibility of the operating system. we can improve the access time and the bandwidth by scheduling the services of disk i/o request in good order.

FCFS scheduling: the simplest form disk scheduling is first come first serve .this algorithm is intrinsically fair but it generally does not provide fast service.
SSTF scheduling: shortest seek time first algorithm services the entire request close to the current position of the head. Before moving the head far away to service other request the SSTF algorithm select the request with the minimum seek time from the current head position.

SCAN scheduling: in the SCAN algorithm, the disk arm starts at one end of the disk and moves forward to other end, servicing request1 as it reaches each cylinder until it gets to other end of the disk.

CSCAN scheduling: circular scan is a variant of scan. Like scan, CSCAN moves the head from one end of the disc to the other service request along the way when the head reaches the other end it immediately return to the beginning of the disk without servicing any other request on the return trip.

ALGORITHMS

Step 1 : Start

Step 2 : Read the number of processes and the requested tracks.

Step 3 : In FCFS, the processes are scheduled according to the order in which they arrive.

Step 4 : In SSTF, the next process to be scheduled is selected as the one requiring the minimum seek time from the current position of disk head.

Step5 : In SCAN, the processes are scheduled from track 0 to highest numbered track. After reaching highest numbered track, it schedules the processes from there to track 0 and so on.

Step 6 : Circular SCAN always schedules the processes from track 0 to the highest numbered track.

Step 7 : Stop.

BANKER'S ALGORITHM

AIM:

To write a program to implement banker's algorithm.

DESCRIPTION

The resource allocation graph algorithm is not applicable to a resource allocation system with multiple instance of each resource type. The deadlock avoidance algorithm which is applicable to such a system is called banker's algorithm. It is less efficient than the resource allocation graph scheme. When a new process enters the system must declare the maximum number of resources in the system. When a member request a set of resource the system must determine whenever the allocation of these resources will leave the system in a state it will allocate the resources. Otherwise the process must wait until some other process release enough resources. Several data structure must be maintained to implement banker's algorithm.

ALGORITHM:

```
Step 1 : Start
Step 2 : Enter the number of resources and then according to this read the
         resources available to each resource type.
Step 3 : Enter the number of processes and according to this and the number of
         Resources read the maximum claim (row->process, column->resource)
Step 4 : Enter the current allocation table to 2D alloc [][].
Step 5 : Compute the availability of each resource by total unit of particular
         resource – column sum of that resource as index in alloc[][] .
Step 6 : If any process need some resource type some unit and the resources are
         available then the resources are allocated to that process.
Step 7 : After the process's processing , all the resources are released and hence
         consist units of resources are increased.
Step 8 : If all processes can process , then the state in safe else not in safe state.
Step 9 : Stop.
```

FLOPPY STATUS DETAILS

AIM:

To write a program to print floppy status details

DESCRIPTION

int statfs (path, statfsful)

statfs returns information about a mounted file system. “path” is the path name of any file within the mounted file system. Statfsful is a variable of structure statfs.

The statfs structure is declared as

```
struct statf's
 {
   long f_type    /*type of file system*/
   long f_bsize  /*optimal transfer block size*/
   long f_block  /* total data block in file system*/
   long f_bfree  /* free blocks in file system*/
   long f_bavail /* free blocks available to superuser*/
   long f_files    /* total file nodes in the filesystem*/
   long f_ffree   /* free file nodes*/
   long f_namelen /* maximum length of file name */
 };
```

ALGORITHM

```
Step 1: Start
Step 2:  A variable of struct is declared.
Step 3:  Floppy status details such as size of the buffer, total blocks, free blocks
         available   to super user, total file nodes, free file nodes and maximum
         length of filename are printed.
Step 4: The type of file system is obtained by calling the function char * file system
         (long n).
Step 5: Stop.
```

READER’S WRITER’S PROBLEM

AIM:

To implement readers-writers problem.

DESCRIPTION

This implementation still allows a stream of readers to enter the critical section until a writer arrives. When a writer process requests access to the shared resource, any subsequent reader process must wait for the writer to gain access to the shared resource and then release it.

ALGORITHM

```
Step 1:  Start
Step 2:Create semaphore variables mutex1, mutex2. readblock =1,
           writeblock=1,writepending with values as 1.
Step 3: Create a thread for the function reader, within the function reader. Perform
           write pending if there is write operation, also perform sem_wait of read block
           and  mutex1, until writing is completed.
Step 4:To perform read operation, firstly increment the read count and then sem_wait
           (write block) also increment the mutex1 and signal the write pending if it
           suspended by sem_wait.
Step 5:After the read operation, readcount decrement by 1 and signals the writeblock
          if it suspends by sem_wait.
Step 6:To perform writer operation create a thread to perform the function write.
Step 7:Increment writecount by1 and obtain the readblock. It blocks on writeblock
           semaphore,  waiting for all readers to clear the critical section.
Step 8: After the write operation decrement the writecount by 1 if (writecount=
            =0) then signals the readblock
Step 9:Stop.
```

MEMORY MANAGEMENT

AIM

To create a simulator for memory management algorithm.

DESCRIPTION

The main memory must accommodate both operating system and various user processes. the memory is usually divided into 2 partitions, one for resident operating system and one for user process. one of the simplest scheme for memory allocation is to divide memory into a number of fixed size partitions. In general there is at any time a set of holes of various sizes scattered throughout the memory. The set of holes is searched to determine which hole is best to allocate.

First Fit

Allocates the first hole that is big enough. Searching can be started either at the beginning or where the previous first fit search end read.

Best Fit

Best Fit allocates the smallest hole that is big enough. We must search the entire list , unless the list is kept ordered by size.

Worst Fit

Allocates to the largest hole.

ALGORITHM

Step 1 : Start

Step 2: Read the memory partitions number and size of each partition.

Step 3 : read the size of memory needed by the process.

Step 4 : If now memory partition is greater than size , print no memory free.

Step 5 : Else read the choice.

Step 6 : In the First Fit , the first memory partition which is greater than size is allocated.

Step 7 : In Best Fit , the memory partition with minimum arr[] size is allocated.

Step 8 : In Worst fit , the largest memory partition is allocated.

Step 9 : Stop.

Cycle II

1. **CHECK IF THE GIVEN RELATION IS IN BCNF OR IN 3NF**

Aim: Write a program to check whether a given relation schema is in BCNF and 3NF

Theory:

A relation schema R, denoted by R (A1, A2,………………An) is made up of a relation name R and a list of attributes A1,A2……………An . R is called name of the relation. The degree of a relation is the number of attributes n of its relation schema.

If a relation has more than one key each is called a **candidate key**. One of the candidate key is designated to be the **primary key**. That is the candidate key whose values are used to identify the tuples in a relation.

An attribute of relation schema R is called a **prime attribute** of R if it is a member of some candidate key of R.

A **super key** of a relation schema R={A1,A2…………..An} is a set of attributes S(R wit a property that no two tuples t1 and t2 in any legal relation state r of R will have t1[s]=t2[S]. A super key SK specifies uniqueness that no two distinct tuples in a state r of R can have the same value for SK. Every relation has at least one default super key –the set of all its attributes.

Algorithm:

Step 1: Enter the attributes, Functional dependencies and key.

Step 2: Tak e each functional dependency . Apply the conditions of BCNF and 3NF.

Step 3: BCNF (Boyce Codd Normal Form)

A relation schema R is in BCNF w.r.t a set F of functional dependencies if for all FDs in F+ of the form α ->β where $\alpha \subseteq R$ and $\beta \subseteq R$

Atleast one of the following conditions hold

α ->β is a trivial functional dependency ($\beta \subseteq \alpha$)
α is a superkey for schema R

Step 4: 3NF (Third Normal Form)

A relation schema R is in 3NF w.r.t a set F of functional dependencies if for all FDs in F+ of the form α ->β where $\alpha \subseteq R$ and $\beta \subseteq R$

Atleast one of the following conditions hold

α ->β is a trivial functional dependency ($\beta \subseteq \alpha$)
α is a superkey for schema R
β is prime

Step 5: If there is at least one FD which doesnot satisfy BCNF or 3NF, then relation schema is not in BCNF or 3NF

Output
The program is executed and verified the results.

2. IMPLEMENTATION OF B-TREE

Aim: To implement B-Tree

Theory:
A search tree of order p is a tree such that each node contains at most p-1 search values and p pointers in order <P1,K1,P2,K2……Pq-1,Kq-1,Pp> where q<=p;each Pi is a pointer to a child node and each Ki is a search value from ordered set of values . All search values are assumed to be unique. A B-tree is a similar to a search tree but it has additional constraints that ensure that tree is always balanced (all leaf nodes are at same level)

Properties of a B-tree :
A B-tree is a rooted tree having the following properties:
1. Every node x has following fields:
    a. n[x], the number of keys currently stored in node x,
    b. n[x] keys themselves ,stored in nondecreasing order, sothat key1[x] <=key2[x] <=….. <=keyn[x][x],
    c. leaf[x],a Boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains n[x]+1 pointers c1[x],c2[x],….,cn[x]+1[x] to its children. Leaf nodes have no children ,so their ci fields are undefined.
3. There are lower and upper bounds on number of keys a node can contain.These bounds can be expressed in terms of a fixed integer t>=2 called 'minimum degree' of B-tree:
    a. Every node other than root must have at least t-1 keys. Every internal node other than root thus has at least t children.If tree is nonempty,root must have at least one key.
    b. Every node can contain at most 2t-1 keys.Therefore ,an internal node can have at most 2r children.We say that a node is 'full' if it contains exactly 2t-1 keys.

The simplest B-tree occurs when t=2. Every internal node that has either 2,3,or 4 children and we have a 2-3-4 tree

Algorithm :

Basic operations on B-trees

A. Searching a B-tree

```
B-TREE –SEARCH (x,k)
1.   i=0
2.   while i<= n[x]-1 and k>keyi[x]
3.       do i=i+1
4.   if i<=n[x]-1 and k=keyi[x]
5.       Then return (x,i)
6.   if leaf [x]
7.       then return NIL
8.       else
9.             return B-TREE –SEARCH(ci[x],k)
```

B. Creating an empty B-tree

```
B-TREE –CREATE(T)
1.  x=ALLOCATE-NODE()
2.  leaf[x]=TRUE
3   n[x]=0
4.  Root[T] =x
```

C. Inserting a key into a B-tree :

```
B-TREE –INSERT(T,k)
1.  r=root[T]
2.  if n[r]=2t-1
3.      then s= ALLOCATE-NODE()
4.            root[T]=s
5.            leaf[s]=FALSE
6.            n[s]=0
7.            c0[s]=r
8.            B-TREE –SPLIT-CHILD(S,0,R)
9.            B-TREE –INSERT-NONFULL(S,K)
10.     else  B-TREE-INSERT-NONFULL(r,k)
```

```
B-TREE –SPLIT-CHILD(x,i,y)
1. z=ALLOCATE-NODE()
2. leaf[z]=leaf[y]
3. n[z]=t-1
4. for j=0 to t-2
5.        do keyj[z]=keyj+t[y]
6. if not leaf[y]
7.        then for j=0 to t-1
8.                do cj[z]=cj+t[y]
9. n[y]=t-1
10. for j=n[x] downto i+1
11.       do cj+1[x]=cj[x]
12.ci+1[x]=z
13.for j=n[x]-1 downto i
14.       do keyj+1[x]=keyj[x]
15. keyi[x]=keyt-1[x]
16. n[x]=n[x]+1
```

```
B-TREE-INSERT-NONFULL(X,K)
1. i=n[x]
2. if leaf[x]
3.        then while i>=1 and k<keyi-1[x]
4.                do keyi[x]=keyi-1[x]
5.                        i=i-1
6.           keyi[x]=k
7.           n[x]=n[x]+1
8.        else  while I>=1 and k<keyi-1[x]
9.                do i=i-1
10.           if n[ci[x]]=2t-1
11.               then B-TREE –SPLIT-CHILD(x,i,ci[x])
12.                       if k>keyi[x]
13.                               then i=i+1
14.          B-TREE –INSERT-NONFULL(ci[x],k)
```

IMPLEMENTATION DETAILS

A B-tree starts with a single root node(which is also a leaf node)at level 0.When a node is full and a new entry is inserted into it that node is split into two nodes at same level and middle entry is moved to parent node along with two pointers to the new split nodes .

Output :
Construct a B-tree and insert the following integers:
4,9,0,50,3,12,89,45,8,55,2,23,14,11,22
Also search for any integer in the B-tree.

3. CONVERT SQL SUBSET INTO RELATIONAL ALGEBRA

Aim: To convert SQL subset into relational algebra

Algorithm:
1. Enter attribute to be selected ,table name and condition
2. Replace AND with '^' and OR with 'v'
3. Use ASCII value of characters ∏ and σ to replace SELECT and FROM respectively.

Output:
Displays SQL statement in Relational Algebraic form.