LAB MANUAL

SUBJECT:

OPERATING SYSTEM WITH UNIX

CLASS:

T.E Computer (Semester VI)

INDEX

Sr.No.	*Topic*
1	File management system calls: Write a program to implement 1. create a file 2. Copy one file to another 3. Linking a file 4. Delete a file.
2	Directory management system calls: Write a program to change directory and print its contents.
3	Parent process – Child process Relationship.
4	Implementing IPC using pipes.
5	Simulation of scheduling algorithms: Write a program to implement the following process scheduling algorithms 1)First Come First Serve 2)Shortest Remaining Job First 3)Round Robin 4)Preemptive Priority Scheduling
6	Implementation of semaphore: Write a program that demonstrates how two processes can share a variable using semaphore.
7	Implementation of shell : Write a 'c' program to implement a shell.
8	Producer – Consumer Problem : Write a program to implement producer consumer problem (Using POSIX semaphores)
9	To implement Banker's algorithm for a multiple resources.
10	Shell scripts.
11	Dinning Philosopher's problem.
12	To study page replacement policies like 1) OPTIMAL 2) LEAST RECENTLY USED(LRU) 3) FIRST-IN-FIRST-OUT

LAB ASSIGNMENT : 1

Title Write a program to implement

1. Create a file
2. Read contents of a file
3. Write to a file
4. Link and unlink a file
5. Copy file
6. Read contents of a file in a reverse order

Objective To study various file management system calls.

References

1. "Operating Systems", William Stallings
2. "The Design of the Unix Operating Systems", Maurice J. Bach
3. "UNIX concepts and Applications" , Sumitabha Das

Pre-requisite Knowledge of

- File manipulations
- Error handling in Unix

Description:

System Calls :

"System calls are functions that a programmer can call to perform the services of the operating system. "

open : system call to open a file :open returns a file descriptor, an integer specifying the position of this open file in the table of open files for the current process .

close : system call to close a file
read : read data from a file opened for reading
write : write data to a file opened for writing
lseek : seek to a specified position in a file (used for random access when reading or writing a file)
link : create a link to a file.
unlink : unlinks a file.

The open() system call :
The prototype is

```
#include       <fcntl.h>
int     open(const char *path,int oflag);
```

The return value is the descriptor of the file

Returns -1 if the file could not be opened.

The first parameter is path name of the file to be opened and the second parameter is the opening mode specified by bitwise ORing one or more of the following values

Value	*Meaning*
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing
O_APPEND	Open at end of file for writing

O_CREAT	Create the file if it doesn't already exist
O_EXCL	If set and O_CREAT set will cause open() to fail if the file already exists
O_TRUNC	Truncate file size to zero if it already exists

close() system call :

The close() system call is used to close files. The prototype is

```
#include      <unistd.h>
int     close(int fildes);
```

It is always a good practice to close files when not needed as open files do consume resources and all normal systems impose a limit on the number of files that a process can hold open.

The read() system call :

The read() system call is used to read data from a file or other object identified by a file descriptor. The prototype is

```
#include      <sys/types.h>
size_t  read(int fildes,void *buf,size_t nbyte);
```

fildes is the descriptor, *buf* is the base address of the memory area into which the data is read and *nbyte* is the maximum amount of data to read.

The return value is the actual amount of data read from the file. The pointer is incremented by the amount of data read.

An attempt to read beyond the end of a file results in a return value of zero.

The write() system call :

The write() system call is used to write data to a file or other object identified by a file descriptor. The prototype is

```
#include      <sys/types.h>
size_t  write(int fildes, const void *buf, size_t nbyte);
```

fildes is the file descriptor, *buf* is the base address of the area of memory that data is copied from, *nbyte* is the amount of data to copy. The return value is the actual amont of data written.

lseek() System call:

Whenever a read() or write() operation is performed on a file, the position in the file at which reading or writing starts is determined by the current value of the **read/write pointer**. The value of the read/write pointer is often called the **offset**. It is always measured in bytes from the start of the file. The lseek() system call allows programs to manipulate this directly so providing the facility for **direct access** to any part of the file.

It has three parameters and the prototype is

```
#include      <sys/types.h>
#include      <unistd.h>
long          lseek(int fildes,off_t offset,int whence)
```


fildes is the file descriptor, *offset* the required new value or alteration to the offset and *whence* has one the three values :-

Symbolic value	***meaning***

SEEK_SET	set pointer to value of *offset*
SEEK_CUR	set the pointer to its current value plus the value of *offset* which may, of course, be negative
SEEK_END	set the pointer to the size of the file plus the value of *offset*

link() System call:

The link() system call is used to create a hard link to an existing file. This means that there will be 2 (or more) directory entries pointing to the same file.

The prototype is

```
#include      <unistd.h>
int     link(const char *existing, const char *new);
```

Where *existing* is the path name of an existing file and *new* is the name of the link to associate with the file. Both parameters can be full path names rather than file names.

Existing as well new refer to the same file and so have same permissions ownerships and inode numbers. On success 0 is returned. On failure 1 is returned.

The unlink() system call :

The unlink() system call removes a directory entry. This will decrement the link count and, if that goes to zero, will cause the file to be deleted.

The prototype is

```
#include      <unistd.h>
int unlink(const char *path);
```

4

In order to *unlink()* a file you require write access to the directory containing the file because the directory contents are going to be changed.

Sample Output :

```
Menu:
1) CREATE
2) READ
3) WRITE
4)  LINK
5)  UNLINK
6) DELETE
7) COPY
8) EXIT

Enter your choice:1
Enter name of the file to be created: demo
File demo is created
Enter your choice : 3
Enter name of the file : demo
Enter text to be entered in demo file : Hello this is a sample file
Enter your choice : 2
Enter name of the file to be read : demo
Contents are : Hello this is sample file.
Enter your choice : 8
```

Conclusion : Various file management system call have been studied successfully.

Post Lab Assignment:

Q-1 What is the difference between library procedure and system calls?

Q-2 what is the difference between the following two commands

1) cat filename 2) cat > filename

Q-3 Write a UNIX command to display file access permissions of a specific file?

Q-4 Explain use of **chmod** command in UNIX.

LAB ASSIGNMENT : 2

Title : Write a program to change current working directory and display the inode details for each file in the new directory.

Objective :: Study various Directory management system calls.

References : 1 . "UNIX Concepts and Applications", Sumitabha Das
2. "The Design of the Unix Operating Systems", Maurice J. Bach

Pre-requisite : Basics of UNIX file and Directory system.

Description:
Inode : Every file is associated with a table called inode table which contains all the static information about the file. It contains following information.

```
struct stat{
        ino_t st_ino;           // Inode number
        mode_t st_mode;         // Type and permissions
        nlink_t st_nlink;       // Number of hard links
        uid_t st_uid;                   // user id
        gid_t st_gid;                   // Group id
        dev_t st_rdev;          // Device ID
        .
        .
        }
```

stat() system call is used to extract inode information.

```
struct stat statbuf;
int stat(const char *path , &statbuf);
```

Extracting File type and permission : The st_mode member of stat combines filetype and permissions. It contains 16 bits.

1- 4 Type
5-7 UID , GID and Sticky bit
8-10 Owner permissions
11-13 Group permissions
14-16 Others permissions

Thus four left most bits represents file type and rest of the 12 bits represent permission. To extract these components separately we need to use S_IFMT mask.

```
mode_t file_type,perm;
file_type = statbuf.st_mode & S_IMFT; // 1-4 nits
perm = statbuf.st_mode & ~S_IMFT; // 5-16 bits
```

UNIX offers a number of system calls to handle a directory.
The following are most commonly used system calls.

1. opendir()

Syntax : DIR * opendir (const char * dirname);
Opendir () takes dirname as the path name and returns a pointer to a DIR structure. On error returns NULL.

2. readdir()

Syntax: struct dirent * readdir (DIR *dp) ;
A directory maintains the inode number and filename for every file in its fold. This function returns a pointer to a dirent structure consisting of inode number and filename.

'dirent' structure is defined in <dirent.h> to provide at least two members – inode number and directory name.

```
struct dirent
{
    ino_t d_ino ;  // directory inode number
    char d_name[]; // directory  name
}
```

3. closedir()

Syntax: int closedir (DIR * dp);
Closes a directory pointed by dp. It returns 0 on success and -1 on error.

4.getcwd ()

Syntax : char * getcwd (char * buff, size_t size);
This function takes two parameters – pointer to a character buffer and second is the size of buffer. The function returns the path name current working directory into a specified buffer.

5. chdir()

Syntax: int chdir (const char * pathname);
Directory can be changed from the parent directory to a new directory whose path is specified in the character buffer pathname.

Sample Output:

```
bash-3.00$ ./a.out /home/staff/archanags/OS/OSP
Old dir : /home/staff/archanags/OS  New dir = /home/staff/archanags/OS/OSP
Type    Inode          name              Type      Permissions
d       6406912        .                 40000     755
d       6211096        ..                40000     755
f       6406913        .abc.swp          100000    640
f       6406158        a.out             100000    755
f       6406214        os1.c             100000    644
f       6406714        os1.c~            100000    644
f       6406918        os10.c            100000    644
```

Conclusion : Various Directory management system calls have been studied successfully.

Post Lab Assignment:

1. Explain INODE.
2. How long can a UNIX filename be and which character can't be used in a filename?
3. What does **cd** do when used without arguments ?

LAB ASSIGNMENT : 3

Title : Parent process – Child process Relationship.

Objective : To study creation of process.

References:

1. "Operating Systems", William Stallings
2. "The Design of the Unix Operating Systems",Maurice J. Bach
3. "Modern Operating Systems",Tannenbaum,

Operating System with UNIX

Eastern Economy edition , 2nd Edition Year 199

Pre-requisite : Knowledge of Process Concept

Description:

The system( ) function is a library function. It is constructed from the system calls execl(), fork() and wait().

The entire process life cycle is built around four system calls.

1. execl()
2. fork()
3. wait()
4. exit()

1. The execl ()

It is used to run a command within a C program.

Syntax :

int execl (path, Arg 0, Arg 1, Arg2, ...,0);

The path argument is the name of the file holding the command we wish to run.

Arg 0 is the name of the command. and Arg1 ,Arg2, Arg3,...are the list of arguments required for that particular command.

Arg 0 is used to mark the end of the argument list.

Example :

```
# include < stdio.h>
main()
{
printf(" Here comes the data : \n")
execl("/bin/date" , "date", 0);
printf ("Hello");
}
```

Here execl() overlays the original program with the called program, and the called program becomes the program of record. In terms of processes, the summoned command has the same ID that the calling program has before it was overlaid. In other words, the process remains the same, but the program instructions in the process are replaced.

2. The fork () system call

fork() is used to create a new process.

The execl () command does not start a new process , it just continues the original process by overlaying memory with a new set of instructions. As this occurs a new program is replaced so there is no way to return to the old program. Some times it is necessary to start a new process , leaving the old process unrelated. This is done with the fork().

Syntax:

pid_t fork(void);

Fork creates an exact replica of parent (calling)process. After fork returns, the parent process and child process now have different PIDs.. At this point , there are two processes with practically identical constitute, and they both continue execution at the statement following fork(). To be able to distinguish between the parent and the child process , fork returns with two values:

Zero in the child process.

The PID of the child in the parent process.

Example :

```
# include <stdio.h>
main()
{
        int pid;
        pid = fork();
        if ( pid == 0)
        {
        // this is the child process;
        }
        else
        {
        // This is the  Parent process.
```

```
        {
}
```

3. The wait () system call

This function blocks the calling process until one of its *child* processes exits.

Syntax :

```
int wait ( int * status);
```

The function returns the process identification number of the terminated processes. Further information concerning the terminated process is written into the location whose address is supplied as the function parameter. One of the main purposes of **wait()** is to wait for completion of child processes.

The execution of **wait()** could have two possible situations.

1. If there are at least one child processes running when the call to **wait()** is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.
2. If there is no child process running when the call to wait() is made, then this wait() has no effect at all.

4. The waitpid() system call

The system call waitpid() like *wait()* suspends the calling process until one of its children changes state. It is possible to specify to wait for a child with a specific process id, wait for a child belonging to a specific process group and wait for any child. It is also possible to specify what type of child process status change is to be waited for.

5. The exit() system call

The system call exit() is used to terminate the current process. The system call wait() is used when a process wishes to wait until a child process terminates and determine the exit code of the child process.

Conclusion : Parent process – Child process relationship has been studied.

Post Lab Assignment:

1. what is the difference between wait() and waitpid() ?
2. What do you mean by zombie state ?
3. How can you distinguish between parent process and child process.
4. What is the difference between fork() and execl() system call ?

LAB ASSIGNMENT : 4

Title : Write a program that creates a child process. Parent process writes data to pipe and child process reads the data from pipe and prints it on the screen.

Objective : To study of Inter Process Communication (IPC) using Pipes.

References :

1. "Operating Systems", William Stallings
2. "The Design of the Unix Operating Systems",Maurice J. Bach
3. "Modern Operating Systems",Tannenbaum,
Eastern Economy edition , 2nd Edition Year 199

Pre-requisite: Knowledge of parent- child process and pipes .

Description:

One of the mechanisms that allow related-processes to communicate is the pipe. A pipe is a one-way mechanism that allows two related processes (i.e. one is an ancestor of the other) to send a byte stream from one of them to the other one.

The system assures us of one thing: The order in which data is written to the pipe, is the same order as that in which data is read from the pipe. The system also assures that data won't get lost in the middle, unless one of the processes (the sender or the receiver) exits prematurely.

The pipe() system call

This system call is used to create a read-write pipe that may later be used to communicate with a process we'll fork off. The call takes as an argument an array of 2 integers that will be used to save the two file descriptors used to access the pipe. The first to read from the pipe, and the second to write to the pipe. Here is how to use this function:

```
int fd[2];
if (pipe(fd) < 0)
 perror("Error");
```

If the call to pipe() succeeded, a pipe will be created, fd[0] will contain the number of its read file descriptor, and fd[1] will contain the number of its write file descriptor.

Our program first call fork() to create a child process. One (the parent process) reads write to the pipe and child process reads the data from the pipe ans then prints the data to the screen.

Conclusion: Inter Process Communication (IPC) using Pipes has been studied.

Post Lab Assignment:

What do you mean by named and unnamed pipes ?

LAB ASSIGNMENT 5 :

Title : To write a program to implement the following process scheduling algorithms

1)First Come First Serve

2)Shortest Remaining Job First

3)Round Robin

4)Pre-emptive Priority Scheduling

References :

1. "Operating Systems Concepts", Silbershatz,Peterson, Galvin,
Addison Wesley 2nd Edition
2. "Modern Operating Systems",Tannenbaum,
Eastern Economy edition , 2nd Edition Year 1995

Pre-requisite :

Knowledge of Scheduling Policies.

Description:

Scheduling algorithms are used when more than one process is executable and the OS has to decide which one to run first.
Terms used
1)Submit time :The process at which the process is given to CPU
2)Burst time : The amount of time each process takes for execution
3)Response time :The difference between the time when the process starts execution and the submit time.
4)Turnaround time :The difference between the time when the process completes execution and the submit time.

First Come First Serve(FCFS)
The processes are executed in the order in which they have been submitted.

Shortest Job First(SJF)
The processes are checked at each arrival time and the process which have the shortest remaining burst time at that moment gets executed first. This is a preemptive & non-preemptive algorithm

Round Robin
Each process is assigned a time interval called its quantum(time slice)
If the process is still running at the end of the quantum the CPU is preempted and given to another process, and this continues in circular fashion, till all the processes are completely executed.

Preemptive Priority Scheduling
Each process is assigned a priority and executable process with highest priority is allowed to run

Conclusion : Various process scheduling algorithms have been studied successfully.

Post Lab Assignment: Comparative Assessment of various Scheduling Policies.

LAB ASSIGNEMENT : 6

Title : Write a program that demonstrates how two processes can share a variable using semaphore.

Objective : Implementation of semaphore.

References : 'The C Odyssey UNIX' by Vijaymukhi.

Pre-requisite : semaphore concepts

Description :
Semaphore : Semaphore is a protected variable that can be accessed and manipulated by means of only two operations.
1) Wait() :
   If (s > 0) then decrements the value of s. otherwise the process is blocked on s.
2) Signal()
   Increments the value of semaphore. If semaphore queue is not empty , it moves one of the blocked processes to the ready queue.

   Binary Semaphore : can take on only two values 0 and 1
   Counting Semaphore : Can take any non-negative integer value.

Use of semaphore :
1) To achieve mutual exclusion
2) For synchronization among processes.

Creating a semaphore :
syntax :
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget (key_t key, int nsems, int semflg);
```

To the variable key we assign a hexadecimal value(0x20). This is the name that we create the semaphore with.
nsem is the number of sub-semaphores in the semaphore set.
semflg can one of the following.
IPC_CREATE | 0666: creates a semaphores with read and alter permission for each group.
IPC_CREAT | IPC_EXCL | 0666 : creates a semaphore in exclusive mode with the the given permissions. If semaphore with the specified key already exists, generates an error.
semget() returns a value which is the identifier of the semaphore . Returns -1 on error.

Getting and Setting semaphore values :
Syntax :
```
int semctl(int semid, int semnum, int cmd, ...);
```
The function semctl performs the control operation specified by cmd on the semaphore set identified by semid, or on the semnum-th semaphore of that set.
semid : semaphore identifier
semnum : is the subsemaphore number whose value we want to get ot set. 0 value refers to the first sub-semaphore in the set.
cmd can be one of the following.
GETVAL : Returns the value of specified sub-semaphore.
GETALL : Returns the values of all sub-semaphores from a specified semaphore set.
SETVAL : sets the value of specified sub-semaphore.
GETPID : Returns the pid of the process that has last altered the semaphore value.
IPC_RMID : Removes the specified sub-semaphore.

GETVAL and SETVAL are not indivisible operations . They do not guarantees mutual exclusion. So instead of GETVAL and SETVAL we sue structures and semop() functions, because they give us the indivisibility we desire.

semop() function:
semop() performs operation on semaphore set.
prototype:
```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```
semid : it is the semaphore id returned by a previous semget() call.
sops : sops argument is a pointer to an arrray of structures of type sembuff.

struct sembuf

```
{
        u_short sem_num; /* semaphore number */
        short sem_op;      /* semaphore operation */
        short sem_flg;  /* operation flags */
};
```

sem_num is the number associated with a sub semaphore. 0 indicates the first sub semaphore.
sem_op is a value that defines the operation we want to performed on the semaphore. Through this value we can define whether we want to capture the resource or release it.
sem_op member can be passed three kinds of values.
a positive integer increments the semaphore value by that amount.
a negative integer decrements the semaphore value by that amount. An attempt to set a semaphore to a value less than 0 fails or blocks.(if sem_flg is set to IPC_NOWAIT).
a value 0 means to wait for the semaphore value to reach 0.
sem_flg can be one of the following:
It defines the step to take , if the semaphore is already in use by another process.
1)IPC_NOWAIT allows the process to carry on with some other task.
2) SEM_UNDO : Allows individual operations in the array of semaphore operations (sop array) to be undone when the process exists.
nsops : Is the number of operations to be performed on the semaphore.

Conclusion : Concept and use of semaphores have been studuied.

Post Lab Assignment:
1) What are the requirements of mutual exclusion ?
2) Explain hardware approaches to mutual exclusion.

LAB ASSIGNMENT : 7

Title : Write a 'c' program to implement a shell.

Objective : Implementation of shell.

References : Internet , The 'C' Oddyssey UNIX by Vijaymukhi.

Pre-requisite : Working of shell.

Description:

The OS **command interpreter** is the program that people interact with in order to launch and control programs. On UNIX systems, the command interpreter is usually called the *shell*: it is a user-level program that gives people a command-line interface to launching, suspending, and killing other programs. sh, ksh, csh, tcsh, bash, ... are all examples of UNIX shells

Every shell is structured as the following loop:

1. print out a prompt
2. read a line of input from the user
3. parse the line into the program name, and an array of parameters
4. use the fork() system call to spawn a new child process
    - the child process then uses the execv() system call to launch the specified program
    - the parent process (the shell) uses the wait() system call to wait for the child to terminate
5. when the child (i.e. the launched program) finishes, the shell repeats the loop by jumping to 1.

We will use following system calls in order to implement UNIX shell.
1) strtok()
#include <string.h>
char *strtok(char *s, const char *delim);
A `token' is a nonempty string of characters not occurring in the string *delim*, followed by \0 or by a character occurring in *delim*.

The **strtok()** function can be used to parse the string *s* (first parameter)into tokens. The first call to **strtok()** should have *s* as its first argument. Subsequent calls should have the first argument set to NULL. Each call returns a pointer to the next token, or NULL when no more tokens are found.

If a token ends with a delimiter, this delimiting character is overwritten with a \0 and a pointer to the next character is saved for the next call to **strtok()**. The delimiter string *delim* may be different for each call.

2) execvp()

Syntax : int execvp (const char *file, char *const argv[]);

The exec family of functions replaces the current process image with a new process image. The functions execvp will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash (/) character. The search path is the path specified in the environment by the PATH variable. If this variable isn't specified, the default path ``/bin:/usr/bin:" is used .

Conclusion : Shell is implemented successfully.

Post Lab Assignments:

1) What do you meant by shell?
2) What are 3 standard system files ? List out file descriptor for them.

LAB ASSIGNMENT : 8

Title : write a program to implement producer consumer problem
(Using POSIX semaphores)

Objective : Solving producer – consumer problem using POSIX semaphore.

References :
1. Unix Network Programming By Richard Steven
2. Modern operating system by Tenenbaum

Pre-requisite : Producer consumer problem

Description :
The producer-consumer problem (Also called the bounded-buffer problem.) illustrates the need for synchronization in systems where many processes share a resource. In the problem, two processes share a fixed-size buffer. One process(producer) produces information and puts it in the buffer, while the other process (consumer) consumes information from the buffer. These processes do not take turns accessing the buffer, they both work concurrently. Herein lies the problem. What happens if the producer tries to put an item into a full buffer? What happens if the consumer tries to take an item from an empty buffer?

In order to synchronize these processes, we will block the producer when the buffer is full, and we will block the consumer when the buffer is empty. So the two processes, Producer and Consumer, should work as follows:

Algorithm :

Assuming there are total N number of slots.

Initialization: semaphores: mutex = 1; Full = 0 , empty = N;

integers : in = 0; out = 0;

```
producer :
        Repeat for ever
                produce (item);
                wait(empty);
                wait(mutex);
                enter_item(item);
                signal( mutex );
                signal( full );
Consumer :
        Repeat forever
                wait(full );
                wait( mutex );
                remove_tem(item);
                signal( mutex );
                signal(empty);
```

POSIX : POSIX stands for Portable Operating System Interface

sem_init()
Initializes a semaphore .

```
#include <semaphore.h>
    int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- First argument is the pointer to semaphore , that you want to initialize. sem_init initializes the semaphore object pointed to by sem.
- The pshared argument indicates whether the semaphore is local to the current process (pshared is zero) or is to be shared between several processes (pshared is not zero). LinuxThreads currently does not support process-shared semaphores, thus sem_init always returns with error ENOSYS if pshared is not zero.
- Third argument is the value of the semaphore. The count associated with the semaphore is set initially to value.

pthread_create()

```
#include <pthread.h>
int pthread_create(pthread_t * thread, pthread_attr_t * attr, void *(*start_routine)(void *), void * arg);
```

pthread_create creates a new thread of control that executes concurrently with the calling thread. The new thread applies the function start_routine passing it arg as first argument. The new thread terminates either explicitly, by calling pthread_exit(3), or implicitly, by returning from the start_routine function. The latter case is equivalent to calling pthread_exit(3) with the result returned by start_routine as exit code.
The attr argument specifies thread attributes to be applied to the new thread.

pthread_join()

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

pthread_join suspends the execution of the calling thread until the thread identified by **th** terminates, either by calling pthread_exit(3) or by being cancelled. If thread_return is not NULL, the return value of th is stored in the location pointed to by thread_return. The return value of th is either the argument it gave to pthread_exit(3), or PTHREAD_CANCELED if th was cancelled.
The joined thread th must be in the joinable state.

Conclusion : Producer consumer problem is solved successfully using POSIX threads and semaphore.

Post Lab Assignment:
1. What is race condition ?
2. What is multi-threading ? write advantages of multi-threading

LAB ASSIGNMENT : 9

Title : To implement Banker's algorithm for a multiple resources.

Objective: Implementation of Banker's Algorithm to avoid deadlock.

References :
1. "Operating Systems", William Stallings
2. "Modern Operating Systems",Tannenbaum, Eastern Economy edition , 2nd Edition Year 1995

Pre-requisite :

Knowledge of Banker's Algorithm.

Description :

This uses a deadlock avoidance policy
Banker's algorithm is applied to arbitrary number of processes and arbitrary number of resource classes each with multiple instance.
The banker's algorithm mainly consists of the following matrices:
i. The resources assigned matrix which shows the number of resources of each type are currently assigned to each process.

ii. Resources still needed matrix indicates how many resources each process need in order to complete. For this the process must state total resources needed before executing the program. It also consists of three vectors.
P vector => Processed resources
A vector => Available resources
E vector => Existing resources

Algorithm to check if a state is safe or not:

i. Look for a row R whose count unmet resources needs are all smaller than A. if no such row exists, the system is deadlocked since no process can run to completion.
ii. Assume the process of the row chosen requests all the resources it needs & finishes. Mark the process as terminated & add it's resources to A vector.
iii. Repeat steps i & ii, until either all processes are marked terminated.

If several processes are eligible to be chosen in step1, it does not matter which one is selected.

Conclusion : Banker's algorithm to avoid deadlock is implemented successfully.

Post Lab Assignment :

1) What are the condition to occur a deadlock?
2) What are the strategies to deal with deadlock?

LAB ASSIGNMENT : 10

Title : Shell scripting.

Objective : To learn shell programming

References : UNIX Shell Programming by Yashwant Kanetkar.

Pre-Requisite : Programming concepts

Description:

Introduction

Shells are interactive, which means that they can accept commands from you via keyboard and execute them. It is possible to store a command in a sequential manner to a text file and tell the shell to execute the file, instead of entering the commands one by one. This is known as a shell script. The shell also incorporate powerful programming language that enables the user to exploit the full power and versatility of UNIX. Shell scripts are powerful tool for invoking and organizing UNIX commands. The shell's more advanced script can do decision sequences and loops thereby performing almost any programming task. A script can provide interaction ,interrupts and error handling and status report just like UNIX commands.

Why to use Shell Script

- Shell Script can take input from user, file and output them onto screen.
- Useful to create our own commands.
- Save lots of time.
- To automate some task of day-to-day.10
- System administration can automated.

The three widely used UNIX shells are Bourne shell , C shell and Korn shell. All the shells supports processes,pipes,directories and other features of UNIX. It may happen that the shell scripts written for one shell may not work on other. This is because the different shells use different mechanism to execute the commands in the shell script.

Basic of Shell script :

Create a file and write all UNIX commands that you want to execute and save that file with extension .sh.

Example : Create a file script1.sh and type following commands in it.

```
ls
who
```

Use the following command to run this script.

```
Bash$ sh script1.sh
```

1) Displaying message on the screen
   echo message
2) reading input in a variable
   read m1,m2,m3
   where m1 m2 and m3 are variables.
3) Display value of a variable
   echo $m1 $m2 $m3
4) Assigning value to a variable
   m1=10 m2=20 m3=30
5) Comments in Shell
   # this is a comment
6) Performing arithmetic operations
   Because all shell variables are string variables. If we want to perform arithmetic operation on these variables we have to use command **expr.**

```
a=20 b=30
echo `expr $a + $b`
echo `expr $a \* $b`
echo `expr $a / $b`
```

7) Control Instructions in shell

- **if-then-fi**

```
if control command
then
    # statements
fi
```

- **if-then-else-fi**

```
if control command
then
    #statements
else
    # statements
fi
```

- using test
  this command translate the result into the language of true and false

```
if test condition
then
    # statements
fi
```

```
e.g.  if test $num -lt 6
      then
            echo "value is less than 6"
      fi
```

-gt : greate than
-lt : less than
-ge : Greater than or equal to
-le : Less than or equal to
-ne : Not equal to
-eq : equal to

- Logical operators:
  -a (AND)

-o (OR)
! (NOT)

- Loops

```
while [condition]
do
    #statements
    done
until [condition]
do
        # statements
done

for counter in *
do
        #statements
done
```

Conclusion : Shell script is studied successfully

Post Lab Assignment:

1) What do you mean by background process ? How to run a process in a background?
2)

LAB ASSIGNMENT : 11

Title : To study Dinning Philosophers Problem.

Objective: To understand How to avoid Starvation

References : 1. "Modern Operating Systems",Tannenbaum,
2. Eastern Economy edition , 2nd Edition Year 1995

Pre-requisite: Concept of semaphore,System calls related to semaphore

Description : Five philosophers are seated around a circular table. Each philosopher has a place of spaghetti and he needs two forks to eat. Between each plate there is a fork. The life of a philosopher consists of alternate period of eating & thinking. When a philosopher gets hungry, he tries to acquire his left fork, if he gets it, it tries to acquire right fork.

In this solution, we check after picking the left fork whether the right fork is available or not. If not, then philosopher puts down the left fork & continues to think. Even this can fail if all the philosophers pick the left fork simultaneously & no right forks available & putting the left fork down again. This repeats & leads to starvation.

Now, we can modify the problem by making the philosopher wait for a random amount of time instead of same time after failing to acquire right hand fork. This will reduce the problem of starvation.

Solution to dinning philosophers problem.

Algorithm :

```
philosopher()
{
        int i;10
        while(true)
        {
          think();
          take_fork(i);
          eat();
          putfork(i);
        }
}
```

```
putfork(i)
{
        int i;
        down(mutex);
        state[i] = THINKING;
  test(left);
        test(right);
        up(mutex);
        test(i);
        if( state[i] == HUNGRY && state[i] != EATING && state[right] != EATING)
        {
            state[i] = EATING;
            up(s[i]);
         }
}
```

```
take_frok(i)
{
    int i;
    down(mutex);
    state[i] = HUNGRY;
    test[i];
    up(mutex);
    down(s[i]);
}10
```

Conclusion: Dining philosopher problem has been solved.

Post Lab Assignment :

What is starvation? How is it different from deadlock? How do you avoid starvation?

LAB ASSIGNMENT : 12

Title : To study page replacement policies like

1) OPTIMAL

2) LEAST RECENTLY USED(LRU)

3) FIRST-IN-FIRST-OU

Objective : To study various system calls

References:

1. "Operating Systems", William Stallings
2. "Operating Systems Concepts", Silbershatz,Peterson, Galvin, Addison Wesley 2nd Edition
3. "Modern Operating Systems",Tannenbaum, Eastern Economy edition , 2nd Edition Year 1995

Pre-requisite: Knowledge of Page Replacement Policies.

Description:

In multiprogramming system using dynamic partitioning there will come a time when all of the processes in the main memory are in a blocked state and there is insufficient memory. To avoid wasting processor time waiting for an active process to become unblocked. The OS will swap one of the process out of the main memory to make room for a new process or for a process in Ready-Suspend state.

Therefore, the OS must choose which process to replace.

Thus, when a page fault occurs, the OS has to change a page to remove from memory to make room for the page that must be brought in. If the page to be removed has been modified while in memory it must be written to disk to bring the disk copy up to date.

Replacement algorithms can affect the system's performance. Following are the three basic page replacement algorithms:

<u>Optimal Page Replacement Policy</u>

The idea is to replace the page that will not be referenced for the longest period of time.

Least Recently Used Algorithm
This paging algorithm selects a page for replacement that has been unused for the longest time.

First-In-First_Out
Replace the page that has been in memory longest, is the policy applied by FIFO. Pages from memory are removed in round-robin fashion. Its advantage is it's simplicity.

Conclusion :
Various page replacement algorithms have been studied successfully.

Post Lab Assignments:
Comparative Assessment of various page replacement policies.