UNIVERSITY OF PUNE

LAB COURSE I

SYSTEM PROGRAMMING AND OPERATING SYSTEM (CS-341)

T.Y.B.SC.(COMPUTER SCIENCE)

SEMESTER II

Name ________________________________ **Roll No.** _______

College ________________________________ **Division** _______

Academic Year _____________

Assignment No. 1:
Topic: Extended Shell

Ready reference:

Objective

The basic objective of this assignment is to create a “Command Interpreter” that behaves like a UNIX / LINUX shell Interpreter.

The program written by the user should be able to execute all the UNIX / LINUX commands and also the LINUX simulated commands.

1. Assignment layout

The assignment has two parts

- **a.** To display a User Shell Prompt.
- **b.** To Execute Commands : again divided into two parts -
    - **i.** To execute LINUX Commands.
    - **ii.** To Simulate LINUX Commands and execute through our program.

2. Basic Linux Commands

Command	**Meaning**
cal month year	Prints a calendar for the specified month of the specified year.
Find path -name pattern -print	Searches the specified path for files with names matching the specified pattern (usually enclosed in single quotes) and prints their names. The find command has many other arguments and functions; see the online documentation.
diff file1 file2	Compares two files, reporting all discrepancies. Similar to the cmp command, though the output format differs.
cmp file1 file2	Compares two files, reporting all discrepancies. Similar to the diff command, though the output format differs.
cat files	Prints the contents of the specified files. Sends file contents to standard output. This is a way to list the contents of short files to the screen. It works well with piping.

more	Allows file contents or piped output to be sent to the screen one page at a time
less	Opposite of the more command
pwd	Print working Directory
cd DirectoryName	Change Directory

Command , Syntax & Meaning	**Options**
mkdir [OPTION] DIRECTORY Create a Directory, if not already exist.	**-m, mode=MODE** set permission mode (as in chmod), not rwxrwxrwx - umask **-p**, parents no error if existing, make parent directories as needed **-v**, verbose print a message for each created directory **-help** display this help and exit **-version** output version information and exit
cd [OPTIONS] DirName Change Directory	-P Do not follow symbolic links -L Follow symbolic links (default)
mv [OPTION] SourdeDir TargetDir Change the name of Directory	-b --backup Make a backup of each file that would otherwise be overwritten or removed. -f --force Remove existing destination files and never prompt the user. -i --interactive Prompt whether to overwrite each existing destination file, regardless of its permissions. If the response does not begin with `y' or `Y', the file is skipped. -S *SUFFIX* --suffix=*SUFFIX*

	Append *SUFFIX* to each backup file made with `-b'. The backup suffix is ~, unless set with SIMPLE_BACKUP_SUFFIX. -u --update Do not move a nondirectory that has an existing destination with the same or newer modification time. -v --verbose Print the name of each file before moving it. -V *METHOD* --version-control=*METHOD*' Change the type of backups made with `-b'. METHOD can be: t, numbered make numbered backups nil, existing numbered if numbered backups exist, simple otherwise never, simple always make simple backups --help display help and exit --version output version information and exit
rm [OPTION] files Deletes files	-d, --directory unlink directory, even if non-empty (super-user only) -f, --force ignore nonexistent files, never prompt -i, --interactive prompt before any removal -r, -R, --recursive remove the contents of directories recursively -v, --verbose explain what is being done

	--help display this help and exit --version output version information and exit
rmdir [OPTION] DirName Remove existing directory	--ignore-fail-on-non-empty Ignore each failure that is solely because the directory is non-empty. -p, --parents Remove explicit parent directories if being emptied --verbose Output a diagnostic for every directory processed --help Display help and exit --version Output version information and exit
chown [OPTION] OWNER[:[GROUP]] FILE **chown [OPTION] :GROUP FILE** **chown [OPTION] --reference=RFILE FILE** Change the owner and/or group of each FILE to OWNER and/or GROUP. With --reference, change the owner and group of each FILE to those of RFILE.	-c, changes like verbose but report only when a change is made -dereference affect the referent of each symbolic link, rather than the symbolic link itself -h, no-dereference affect each symbolic link instead of any referenced file (useful only on systems that can change the ownership of a symlink) -from=CURRENT_OWNER:CURRENT_GROUP change the owner and/or group of each file only if its current owner and/or group match those specified here. Either may be omitted, in which case a match is not required for the omitted attribute. -no-preserve-root do not treat `/' specially (the default) -preserve-root fail to operate recursively on `/' -f, -silent, -quiet suppress most error messages -reference=RFILE use RFILE's owner and group rather than the specifying OWNER:GROUP values -R, -recursive operate on files and directories recursively -v, -verbose output a diagnostic for every file processed The following options modify how a hierarchy is traversed when the -R option is also specified. If more than one is specified, only the final one takes effect. -H if a command line argument is a symbolic link to a directory, traverse it

	-L traverse every symbolic link to a directory encountered -P do not traverse any symbolic links (default)
chmod [-r] permissions filenames Change file access permissions	r Change the permission on files that are in the subdirectories of the directory that you are currently in. permission Specifies the rights that are being granted. Below is the different rights that you can grant in an alpha numeric format.filenames File or directory that you are associating the rights with Permissions u - User who owns the file. g - Group that owns the file. o - Other. a - All. r - Read the file. w - Write or edit the file. x - Execute or run the file as a program. Numeric Permissions: CHMOD can also to attributed by using Numeric Permissions: 400 read by owner 040 read by group 004 read by anybody (other) 200 write by owner 020 write by group 002 write by anybody 100 execute by owner 010 execute by group 001 execute by anybody
ls [OPTION] Short listing of directory contents	-a list hidden files -d list the name of the current directory -F show directories with a trailing '/' executable files with a trailing '*' -g show group ownership of file in long listing -i print the inode number of each file -l long listing giving details about files and directories -R list all subdirectories encountered -t sort by time modified instead of name

cp sourcefile destinationfile Copy source file to destination file	cp -i myfile yourfile : prompt before overwrite cp -i /data/myfile : Copy the file "/data/myfile" to the current working directory and name it "myfile". Prompt before overwriting the file. cp -dpr srcdir destdir Copy all files from the directory "srcdir" to the directory "destdir" preserving links (-poption), file attributes (-p option), and copy recursively (-r option). With these options, a directory and all it contents can be copied to another dir
ln sourcefile destinationfile Creates a link of sourcefile to destinationfile	ln –s test symlink Creates a symbolic link named symlink that points to the file test Typing "ls -i test symlink" will show the two files are different with different inodes. Typing "ls -l test symlink" will show that symlink points to the file test.
wc [OPTION] filename Print byte, word, and line counts	-w : count number of words in file -c : Count number of characters in a file -l : Count number of lines in a file

3. System Calls used

a. fork ( ) :

System call *fork()* is used to create processes. It takes no arguments and returns a process ID. The purpose of *fork()* is to create a *new* process, which becomes the *child* process of the caller. After a new child process is created, *both* processes will execute the next instruction following the *fork()* system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of *fork()*:

- If *fork()* returns a negative value, the creation of a child process was unsuccessful.
- *fork()* returns a zero to the newly created child process.

- *fork()* returns a positive value, the *process ID* of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function *getpid()* to retrieve the process ID assigned to this process.

Therefore, after the system call to *fork()*, a simple test can tell which process is the child. Please note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.

b. execlp ( ) :

int execlp(const char ****file*, const char ****arg***, ...);**

Replaces the current process image with a new process image.
The initial argument for these functions is the pathname of a file which is to be executed.
The *const char *arg* and subsequent ellipses in the **execl**, **execlp**, and **execle** functions can be thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments *must* be terminated by a **NULL** pointer.
If the **execlp** function returns, an error will have occurred. The return value is -1, and the global variable *errno* will be set to indicate the error.

c. waitpid ( ) : Wait for Specific Child Process.

The waitpid() function allows the calling thread to obtain status information for one of its child processes. The calling thread suspends processing until status information is available for the specified child process, if the *options* argument is 0. A suspended waitpid() function call can be interrupted by the delivery of a signal whose action is either to run a signal-catching function or to terminate the process. When waitpid() is successful, status information about how the child process ended is stored in the location specified by stat_loc.
Parameters
pid

(Input) A process ID or a process group ID to identify the child process or processes on which waitpid() should operate.
stat_loc
(Input) Pointer to an area where status information about how the child process ended is to be placed.
options
(Input) An integer field containing flags that define how waitpid() should operate.

The pid argument specifies a set of child processes for which status is requested. The waitpid() function only returns the status of a child process from the following set:
- If pid is equal to (pid_t)-1, status is requested for any child process. In this respect, waitpid() is then equivalent to wait().
- If pid is greater than (pid_t)0, it specifies the process ID of a single child process for which status is requested.
- If pid is (pid_t)0, status is requested for any child process whose process group ID is equal to that of the calling thread.
- If pid is less than (pid_t)-1, status is requested for any child process whose process group ID is equal to the absolute value of pid.

value	waitpid() was successful. The value returned indicates the process ID of the child process whose status information was recorded in the storage pointed to by *stat_loc*.
0	WNOHANG was specified on the *options* parameter, but no child process was immediately available.
-*1*	waitpid() was not successful. The *errno* value is set to indicate the error.

d. open ( ) :

int open(const char *pathname, int flags);int open(const char *pathname,int flags, mode_t mode);

Given a *pathname* for a file, open() returns a file descriptor.

The file offset is set to the beginning of the file.

A call to open() creates a new *open file description*, an entry in the system-wide table of open files. This entry records the file offset and the file status flags. A file descriptor is a reference to one of these entries; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file. The new open file description is initially not shared with any other process, but sharing may arise via *fork*.

The parameter *flags* must include one of the following *access modes*: O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file read-only, write-only, or read/write, respectively.

open( ) returnS the new file descriptor, or -1 if an error occurred (in which case, *errno* is set appropriately).

e. close ( ) :

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

close( ) closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).
If *fd* is the last copy of a particular file descriptor the resources associated with it are freed; if the descriptor was the last reference to a file which has been removed using *unlink* the file is deleted. close( ) returns zero on success. On error, -1 is returned, and *errno* is set appropriately.

f. read ( ) :

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.
If *count* is zero, **read**() returns zero and has no other results. If *count* is greater than SSIZE_MAX, the result is unspecified.

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because **read**() was interrupted by a signal. On error, -1 is returned, and *errno* is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

g. write ( ) :

#include <unistd.h>
ssize_t write(int *fd*, const void **buf*, size_t *count*);

write() writes up to *count* bytes to the file referenced by the file descriptor *fd* from the buffer starting at *buf*. POSIX requires that a read() which can be proved to occur after a write() has returned returns the new data.

On success, the number of bytes written are returned (zero indicates nothing was written). On error, -1 is returned, and *errno* is set appropriately. If *count* is zero and the file descriptor refers to a regular file, 0 may be returned, or an error could be detected. For a special file, the results are not portable.

h. lseek ( ) :

#include <sys/types.h>
#include <unistd.h>

off_t lseek(int *fildes*, off_t *offset*, int *whence*);

The lseek() function repositions the offset of the open file associated with the file descriptor *fildes* to the argument *offset* according to the directive *whence* as follows:

SEEK_SET
The offset is set to *offset* bytes.

SEEK_CUR

The offset is set to its current location plus *offset* bytes.

SEEK_END

The offset is set to the size of the file plus *offset* bytes.

The lseek() function allows the file offset to be set beyond the end of the file (but this does not change the size of the file). If data is later written at this point, subsequent reads of the data in the gap (a "hole") return null bytes ('\0') until data is actually written into the gap.

Upon successful completion, lseek() returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of *(off_t)-1* is returned and *errno* is set to indicate the error.

i. opendir ( ) :

resource opendir (string *$path* [, resource *$context*])

Opens up a directory handle to be used in subsequent closedir(), readdir(), and rewinddir() calls.

Path : The directory path that is to be opened
Context : For a description of the *context* parameter, refer to the streams section of the manual.

Returns a directory handle resource on success, or FALSE on failure.

If *path* is not a valid directory or the directory can not be opened due to permission restrictions or filesystem errors, opendir() returns FALSE and generates a PHP error of level E_WARNING. You can suppress the error output of opendir() by prepending '@' to the front of the function name.

j. closedir ( ) :

```
#include <sys/types.h>
#include <dirent.h>

int closedir(DIR *dirp);
```

The closedir() function closes the directory stream associated with dirp. A successful call to closedir() also closes the underlying file descriptor associated with dirp. The directory stream descriptor dirp is not available after this call.

k. readdir ( ) :

#include <unistd.h>
#include <linux/dirent.h>
#include <linux/unistd.h>

int readdir(unsigned int *fd*, struct dirent **dirp*, unsigned int *count*);

readdir reads one *dirent* structure from the directory pointed at by *fd* into the memory area pointed to by *dirp*. The parameter *count* is ignored; at most one dirent structure is read.
The *dirent* structure is declared as follows:

```
struct dirent
{
    long d_ino;                  /* inode number */
    off_t 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) */
}
```

d_ino is an inode number. *d_off* is the distance from the start of the directory to this *dirent*. *d_reclen* is the size of *d_name*, not counting the null terminator. *d_name* is a null-terminated file name.
On success, 1 is returned. On end of directory, 0 is returned. On error, -1 is returned, and *errno* is set appropriately.

l. stat ( ) :

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char **path*, struct stat **buf*);

ThIs function returns information about a file. No permissions are required on the file itself, but -- in the case of stat() -- execute (search) permission is required on all of the directories in *path* that lead to the file. stat() stats the file pointed to by *path* and fills in *buf*.
This system call returns a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

The *st_dev* field describes the device on which this file resides.

The *st_rdev* field describes the device that this file (inode) represents.

The *st_size* field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symlink is the length of the pathname it contains, without a trailing null byte.

The *st_blocks* field indicates the number of blocks allocated to the file, 512-byte units. (This may be smaller than *st_size*/512, for example, when the file has holes.)

The *st_blksize* field gives the "preferred" blocksize for efficient file system I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

Some file system types allow mounting in such a way that file accesses do not cause an update of the *st_atime* field.

The field *st_atime* is changed by file accesses

The field *st_mtime* is changed by file modifications. Moreover, *st_mtime* of a directory is changed by the creation or deletion of files in that directory. The *st_mtime* field is *not* changed for changes in owner, group, hard link count, or mode.

The field *st_ctime* is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

4. C Libraries to be included :

a. <stdio.h>
b. <string.h>
c. <sys/types.h>
d. <sys.wait.h>
e. <sys/stat.h>
f. <dirent.h>
g. <fcntl.h>
h. <unistd.h>
i. <stdlib.h>

5. Program Logic

Internally the two parts of the assignment as explained above work as follows :

a. Parent Process :

- Parent Process will display the Shell Prompt.
- Parent Process will accept the command on the prompt displayed.
- Parent Separates the tokens.

b. Child Process :

- Child process will be created after separating the tokens by the parent.
- If child process gets created successfully, then it executes the simulated commnads or it executes the

LINUX commands depending upon the values of the tokens received by child process.

- If user wants to terminate the shell command, the child process will get killed, parnet process is still alive.
- For this the parent always wait for child response.
- If child process does not get created successfully, and the user wants to exit the program, the parent process is terminated.

6. Algorithm

A . Main Program :

a. Display the Shell Prompt (say "MyShell $")

b. Accept a command from user.

c. Separate the command in 4 tokens. (*strtok ( )* function can be used)

- **i.** Token 1 : actual command text.
- **ii.** Token 2 , Token 3 and Token 3 will represent the parameters to the command.
- **iii.** If no parameters are passed, the Tokens 2, 3 & 4 will contain NULL value.

d. Create a child Process by *fork ( )* system call.

- **i.** *Fork ( )* system call returns 0 to the newly created process created successfully.
- **ii.** *Fork ( )* system call returns a positive value i.e. **PID** of the newly created process to Parent
- **iii.** *Fork ( )* system call returns negative value for unsuccessful creation of the child process.

e. If *PID = 0* then the child has got created and the control is now in child process.
At this point there are three ways in the algorithm

- **i.** To execute Simulated commands
  - Check the Token1 for the Simulated Command Text. If matching, Call respective function call.
  - If Token 1 is "Exit" – Terminate child process.
- **ii.** To execute LINUX commands directly.
  - If Token 1 doesnot match with any of the simulated commands, then assume that the

commond on the prompt is LINUX command and execute it with *execlp ( )* System Call.

- In this again we have to check if the command contains the parameters by checking Token 2, 3 & 4 values, if they are non NULL.

iii. To display Error

- The **execlp ( )** system call used above never returns if gets executed successfully.
- If it returns, indicates that there is error in execution of the command. It returns -1 value.
- In our program, after execution of the program, the child process gets terminated directly which executed *execlp ( )*. So the prompt is displayed.
- In case the *execlp ( )* does not get executed and it returns it will ber the immediate next statement where the control comes. This is the point where we can write our error message.

f. If *PID != 0* then child is not present and now parent can decide if it wants to terminate as per user's command, if exit.

Meanwhile the parent loops performing calls to *waitpid*() to monitor the
status of the child. The loop terminates when child termination is detected.

B. Counting of Characters, Words and Lines:

i. If Token 1 is "Count" process this -
ii. Open file present in Token 3 in READONLY mode. (OPEN system call)
iii. Read the file character by character till the end. (READ system call)
iv. Check the character and count the number of characters, words & lines.
v. Depending on the value of Token 2, display the result.

C. Printing Lines :

i. If Token 1 is "Typeline" process this –

ii. Open file present in Token 3 in READONLY mode. (OPEN system call)
iii. If Token 2 is ‘A’ print whole file. (READ system call)
iv. If Token 2 is positive, print ‘N’ lines from the file. (READ system call)
v. If Token 2 is negative, Set file pointer to end of file and read & print ‘N’ lines. (READ & LSEEK system call)

D. Searching the Pattern :

i. If Token 1 is “Search” process this –
ii. Open file present in Token 4 in READONLY mode (OPEN system call)
iii. If Token 2 is ‘F’ , Read one line from the file and search the pattern in file if found print the line and exit.
iv. Repeat the process if not found till the end of file and report error message.

E. Listing the Directory Contents :

i. If Token 1 is “List” process this –
ii. Open the directory in Token 3. (OPENDIR system call)
iii. If Token 2 is ‘F’ , read the filestatus in one ‘stat’ structure and print filenames. (READDIR System call , predefined structures used are dirent & stat)
iv. If Token 2 is ‘N; , count the entries in directory till the READDIR doesnot return NULL. (READDIR system call, Predefined structures used are dirent & stat).
v. If Token 2 is ‘I’, print all the filenames and respective Inodes from the current directly. (READDIR system call, Predefined structures used are dirent & stat).

SET A

1. Write a program that behaves like a shell (Command Interpreter). It has its own prompt say “MyShell $”. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command.

(Hint : System Calls Used : fork ( ) , execlp ( ) , waitpid ( ))

2. Write a program that behaves like a shell (Command Interpreter). It has its own prompt say “MyShell $”. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command :

    a. Count C < filename > : To print number of Characters in the file.
    b. Count W < filename > : To print number of Words in the file.
    c. Count L < filename > : To print number of Lines in the file.

(Hint : System Calls Used : fork ( ) , execlp ( ) , waitpid ( ), open ( ), close( ), read ( ), write ( ))

3. Write a program that behaves like a shell (Command Interpreter). It has its own prompt say “MyShell $”. It should interpret the following command :

    a. Count CW < filename > : To print number of Characters & Words in the file.
    b. Count WL < filename > : To print number of Words & Lines in the file.
    c. Count CL < filename > : To print number of Characters & Lines in the file.
    d. Count CWL < filename > : To print number of Characters, Words & Lines in the file.

(Hint : System Calls Used : fork ( ) , execlp ( ) , waitpid ( ), open ( ), close( ), read ( ), write ( ))

SET B

1. Write a program that behaves like a shell (Command Interpreter). It has its own prompt say “MyShell $ “. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command :

    a. Typeline +n < filename > : To print first ‘n’ lines of file file.

b. Typeline –n < filename > : To print last ‘n’ lines of the file.
c. Typeline a < filename > : To print all lines of the file.

(Hint : System Calls Used : fork ( ) , execlp ( ) , waitpid ( ), open ( ), close( ), read ( ), write ( ), lseek ( ))

2. Write a program that behaves like a shell (Command Interpreter). It has its own prompt say “MyShell $ “. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command :

   a. Search F < pattern > < filename > : To search first occurance of pattern in the file.
   b. Search C < pattern > < filename > : To count number of occurances of pattern in the file.
   c. Search A < pattern > < filename > : To Search number of occurances of pattern in the file.

(Hint : System Calls Used : fork ( ) , execlp ( ) , waitpid ( ), open ( ), close( ), read ( ), write ( ))

3. Write a program that behaves like a shell (Command Interpreter). It has its own prompt say “MyShell $ “. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command :

   a. Typeline P +n < filename > : To print ‘n’ forward lines of file file from the position ‘P’.
   b. Typeline P –n < filename > : To print backward ‘n’ lines of the file from the position ‘P’
   c. Typeline R < filename > : To print the whole file from last lines to first (Reverse).

(Hint : System Calls Used : fork ( ) , execlp ( ) , waitpid ( ), open ( ), close( ), read ( ), write ( ), lseek ( ))

SET C

1. Write a program that behaves like a shell (Command Interpreter). It has its own prompt say “MyShell $ “. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command :

   a. List F < dirname > : To print names of all files in the current directory.
   b. List N < dirname > : To print number of all entries in the current directory.
   c. List I < dirname > : Print names and Inodes of the files in the current directory.

(Hint : System Calls Used : fork ( ) , execlp ( ) , waitpid ( ), open ( ), close( ), read ( ), write ( ), stat ( ), opendir ( ), closedir ( ), readdir ( ))

2. Write a program that behaves like a shell (Command Interpreter). It has its own prompt say “MyShell $ “. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command :

   a. Search N F < pattern > < filename > : To search Nth occurance of pattern in the file, reading the file from start.
   b. Search –N F < pattern > < filename > : To search Nth occurance of pattern in the file, reading the file from end.

(Hint : System Calls Used : fork ( ) , execlp ( ) , waitpid ( ), open ( ), close( ), read ( ), write ( ), lseek ( ))

Assignment Evaluation

0:Not Done [ ]	1:Incomplete [ ]	2.Late Complete [ ]
3:Needs Improvement [ ]	4:Complete [ ]	5:WellDone [ ]

Signature of the Instructor   Date of Completion

Assignment: 2

Topic: CPU Scheduling

Ready reference

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state.
2. Switches from running to ready state.
3. Switches from waiting to ready.
4. Terminates.

- Scheduling under 1 and 4 is non preemptive.
- All other scheduling is preemptive

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – Number of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Optimization Criteria

- Max CPU utilization
- Max throughput

- Min turnaround time
- Min waiting time
- Min response time

There are 5 CPU Scheduling Algorithms:

1. **FCFS**

- Since context switches only occur upon process termination, and no reorganization of the process queue is required, scheduling overhead is minimal.
- Throughput can be low, since long processes can hog the CPU
- Turnaround time, waiting time and response time can be low for the same reasons above
- No prioritization occurs, thus this system has trouble meeting process deadlines.
- The lack of prioritization does permit every process to eventually complete, hence no starvation.

2. **SJF**

With this strategy the scheduler arranges processes with the least estimated processing time remaining to be next in the queue. This requires advanced knowledge or estimations about the time required for a process to complete.

* If a shorter process arrives during another process' execution, the currently running process may be interrupted, dividing that process into two separate computing blocks. This creates excess overhead through additional context switching. The scheduler must

also place each incoming process into a specific place in the queue, creating additional overhead.

* This algorithm is designed for maximum throughput in most scenarios.

* Waiting time and response time increase as the process' computational requirements increase. Since turnaround time is based on waiting time plus processing time, longer processes are significantly affected by this. Overall waiting time is smaller than FIFO, however since no process has to wait for the termination of the longest process.

* No particular attention is given to deadlines, the programmer can only attempt to make processes with deadlines as short as possible.

* Starvation is possible, especially in a busy system with many small processes being run.

3. **Priority**

The O/S assigns a fixed priority rank to every process, and the scheduler arranges the processes in the ready queue in order of their priority. Lower priority processes get interrupted by incoming higher priority processes.

* Overhead is not minimal, nor is it significant.

* Waiting time and response time depend on the priority of the process. Higher priority processes have smaller waiting and response times.

* Deadlines can be met by giving processes with deadlines a higher priority.

* Starvation of lower priority processes is possible with large amounts of high priority processes queuing for CPU time.

4. **Round Robin**

   The scheduler assigns a fixed time unit per process, and cycles through them.

   * RR scheduling involves extensive overhead, especially with a small time unit.

   * Balanced throughput between FCFS and SJN, shorter jobs are completed faster than in FCFS and longer processes are completed faster than in SJN.

   * Fastest average response time, waiting time is dependent on number of processes, and not average process length.

   * Because of high waiting times, deadlines are rarely met in a pure RR system.

   * Starvation can never occur, since no priority is given. Order of time unit allocation is based upon process arrival time, similar to FCFS.

5. **Round Robin with Multilevel Feedback Queues**

This is used for situations in which processes are easily classified into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs.

SET-A

1. Write the simulation program using FCFS. The arrival time and first CPU bursts of different jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU burst should be generated using random function. The output should give the Gantt chart, Turnaround Time and Waiting time for each process and average times.
2. Write the simulation program using SJF(non-preemptive). The arrival time and first CPU bursts of different jobs should be input to the system. The Assume the fixed I/O waiting time (2 units).The next CPU burst should be generated using random function. The output should give the Gantt chart, Turnaround Time and Waiting time for each process and average times.

SET-B

1. Write the simulation program for preemptive scheduling algorithm using SJF/Priority. The arrival time and first CPU bursts of different jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU burst should be generated using random function. The output should give the Gantt chart, Turnaround Time and Waiting time for each process and average times.
2. Write the simulation program for preemptive scheduling algorithm using Round Robin with time quantum of 2 units. The arrival time and first CPU bursts of different jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU burst should be generated using random function. The output

should give the Gantt Chart, Turnaround Time and Waiting time for each process and average times.

SET-C

Write the simulation program using Round Robin with multilevel feedback queues. The arrival time and first CPU bursts of different jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU burst should be generated using random function. The output should give the Gantt Chart, Turnaround Time and Waiting time for each process and average times.

Assignment Evaluation

0:Not Done [ ]   1:Incomplete [ ]   2.Late Complete [ ]
3:Needs Improvement [ ]   4:Complete [ ]   5:WellDone [ ]

Signature of the Instructor   Date of Completion

Assignment No: -3

Title : Deadlock- Banker's Algorithm

Ready Refernce :

Definition:

Deadlock is a situation where each process in a set of processes wait for the resources held by another process. Thus, a set of processes is in a
deadlock state when every process in the set is waiting for an event that can be caused by another process in the set.

Necessary conditions to occur a deadlock:

A deadlock situation can arise if the following four conditions hold simultaneously in the system:

a) Mutual Exclusion : At least one resource must be held in a non sharable mode. Only one process at a time can use the resource.

b) Hold and Wait : A resource must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

c) No Preemption : Resources can not be preempted; that is, a resource can not be released only voluntarily by the process holding it, after that process has completed its task.

d) Circular Wait : A set {P0,P1,...Pn} of waiting processes must exist such that P0 is waiting for a resource that is held by P1,P1 is waiting for a resource that is held by P2,...,Pn-1 is waiting for a resource that is held by Pn and Pn is waiting for a resource that is held by P0.

There are three ways to handle a deadlock:

a) Deadlock Prevention : It is a method to ensure that at least one of the necessary conditions cannot hold.

b) Deadlock Avoidance : This method requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional information, we can decide for each request whether or not a process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently to each process, and the future requests and releases of each process.

c) Deadlock Detection and Recovery : If a system does not employ deadlock prevention or deadlock avoidance algorithm, then a deadlock situation may occur. In this situation, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

Explaination of avoidance algorithm,safe state,safe sequence.
Deadlock Avoidance:

If detail information about the processes and resources is available then
it is possible to avoid deadlock, e.g. which process will require which resources, possibly in what sequence, etc. This information may help to decide the sequence in which the processes can be executed to avoid deadlock. Each request can be analyzed on the basis of number of resources currently available, currently allocated and future requests which may come from other processes. From this information system can decide whether a process should wait or not.
The Deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait can never exist. The resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.
If a safe sequence in which processes can be executed is available, then only we can call the system to be in safe state. A safe sequence is the sequence of process such that, if processes are executed in safe sequence then each process in the sequence will be executed serially, with current available resources. Then release, all the resources held by it making available for next process in the sequence so on. A system without safe sequence is unsafe. Unsafe system may or may not have a deadlock.

Banker's Algorithm is one of the Deadlock avoidance algorithms.

For these we require following data structures:
1. **Allocation** : An n*m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j]=k,then process Pi is currently allocated k instances of resource type Rj.
2. **Max** : An n*m matrix defines the maximum demand of each process. If Max[i][j]=k, then process Pi may request at most k instances of resource type Rj.
3. **Available :** a vector of length m indicates the number of available resources of each type. If Available[j]=k,then there are k instances of resource type Rj available.
4. **Need :** An n*m matrix indicates the remaining resource need of each process.
If Need[i][j]=k,then process Pi may need k more instances of resource type Rj to complete its task. Note that

Need[i][j]=Max[i][j]-Allocation[i][j].

Safety Algorithm:-

This is the algorithm used for finding out whether or not a system is in a safe
state or not.

The algorithm is as follows:

Let n be the number of processes in the system.

Let m be the number of resources

1) Let Work and Finish be the vectors of length m and n respectively.
 Initialize Work = Available and Finish[i]=false for i= 1,2,3,...,n.

2) Find an i such that
   a) Finish[i]=false
   b) Needi <= Work
  If no such i exists, go to step 4.

3) Work = Work + Allocationi
   Finish[i]=true
   Go to step 2.

4) If Finish[i]=true for all i, then the system is in a safe state.

When a request of resources is made by a process the following algorithm is used.

Resource-Request Algorithm:-

Let requesti be the vector for process Pi. If requesti[j]=k,then process Pi wants k instances of resource ttype Rj.

When a request of resources is made by a process the following actions are taken:

1) If requesti <= Needi, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maqximum claim.

2) If requesti <= Available, go to step 3. Otherwise, Pi must wait, since the
   resources are not available.

3) Assume that the system pretend to have allocated the requested resources to the process Pi by modifying the state as follows:
   a) Available = Available - Requesti
   b) Allocationi = Allocationi + Requesti
   c) Needi = Needi - Requesti

   If the resulting resource-allocation state is safe, the transaction is completed and process Pi is allocated its resources i.e the request is granted immediately. If the resulting resource-allocation state is unsafe, then the process Pi must wait for Requesti and the old transaction i.e. assumed resource- allocation state is restored. That is because the system is not in safe state, so request is not granted immediately.

Set A :
Q.1) Consider the following snapshot of a system:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

1) Display the contents of Need array.
2) Check whether the system is in safe state or not. If yes, give the safe sequence.

Q.2) Consider the following snapshot of a system:

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0
P1	1	0	0	0	1	7	5	0				
P2	1	3	5	4	2	3	5	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

1) Display the contents of Need array.
2) Check whether the system is in safe state or not. If yes, give the safe sequence.

Set B :
Q.1) Consider the following snapshot of a system:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

1) Display the contents of Need array.
2) Check whether the system is in safe state or not. If yes, give the safe sequence.
3)If a request from process P1 arrives for (1,0,2) can it be immediately granted?

Q.2) Consider the following snapshot of a system:

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0
P1	1	0	0	0	1	7	5	0				
P2	1	3	5	4	2	3	5	6				

P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

1) Display the contents of Need array.
2) Check whether the system is in safe state or not. If yes, give the safe sequence.
3)If a request from process P1 arrives for (0,4,2,0) can it be immediately granted?

Q.3) Consider the following snapshot of a system:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	1	0	0	0	0
P1	2	0	0	4	0	2			
P2	3	0	3	3	0	3			
P3	2	1	1	3	1	1			
P4	0	0	2	0	0	1			

1) Display the contents of Need array.
2) Check whether the system is in safe state or not. If yes, give the safe sequence.
3) If a request from process P4 arrives for (0,0,1) can it be immediately granted?

Assignment Evaluation
0:Not Done [ ] 1:Incomplete [ ] 2.Late Complete [ ]
3:Needs Improvement [ ] 4:Complete [ ] 5:WellDone [ ]

Signature of the Instructor Date of Completion

Assignment No. :4

Topic: Paging

Ready reference:

Basic memory management concept:

1) Main memory is a central part of the computer system.CPU and I/O system interact with memory.
2) Logical address is the address generated by CPU or programs.
3) Physical address is the address where actual program data are stored in memory.
4) **Virtual memory** is something that appears to exist, but actually does not exist. Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available. If memory requirement of a program is larger than available memory then virtual memory is used.
5) **Paging** is a non-contiguous memory management scheme. User program will allocate a memory wherever available. In the paging memory-management scheme, the operating system retrieves data from secondary storage in same-size blocks called **pages**. The main advantage of paging is that it allows the physical address space of a process to be noncontiguous.
6) Logical memory is the user's memory .It is divided into number of equal units called Logical memory is the user's memory .It is divided into number of equal units called **pages.**
7) Physical memory is divided into number of equal units called **frames.**
8) **Page table** is used to map a page on a frame.

9) **Demand paging** –A page is demanded or requested by a user program, if not present in the memory i.e. in frame .Then it is given from secondary storage device.

10)When a user program tries to access the page which is not present in memory, The situation is called as **page fault.**

Page replacement:

Memory management is one of the important responsibility of the operating system. In program execution it may happen that there is no free frame available then a page fault occurs, System cannot terminate the user program, because of virtual memory.

To bring user page into the frame, a frame is freed and user page brought in. The process of freeing the frame is called as **page replacement.**

The important part is that which frame should be freed. This frame is called as **victim frame** and the page present on that frame is called **victim page.**

<u>Steps involved in page replacement are:</u>

1. Find the location of the desired page on the disk.
2. Find a free frame:
   a. If there is a free frame, use it.
   b. If there is no free frame, use a <u>page replacement algorithm</u>
   to select a victim frame.
   c. write the victim frame to the disk; change the page and frame
   tables accordingly.

3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.

Page Replacement Algorithm:

O. S. uses one of the page replacement schemes lesser the page fault rate, better the page replacement algorithm.

The various page replacement algorithm are:

1. FIFO
2. Optimal Replacement
3. LRU
4. LRU approximation using reference bit/bits
5. MRU
6. Second chance algorithm
7. LFU
8. MFU

Input of page replacement algorithm is **reference string** reference string is stored page numbers of demand pages and next input is the no of page frames available & its main function is to load required page in free frame.

If there is no free frame then it select victim page.

Generation of reference string as follows-

Consider a address sequence 0100,0432,0101,0612,0611,0105....

At 100 bytes per page, this sequence is reduced to the following reference string

1,4,1,6,1,6,1,.......

FIFO (First In First Out):-

In this algorithms all available frames are given to the pages from reference string serially. I. e. first frame to first pass, second frame to second page & so on.

All available/free frames are over at that time, the first frame is selected as victim frame, next time second frame is selected and so on.

<u>Note: If page is not present in frame then page fault is counted.</u>

Data Structure-

1. M: total no of reference string.
2. RS: Stores reference string i.e. demand page number.
3. F: available n free frames.
4. Rear: page is stored at rear frame after insertion rear is increments by one.
5. Front: It always points to first page after deletion it is increment by one.

Algorithm:

- Find out first free frame.
- If free frame is not available then free front frame and update rear and Front
- Load F[rear] =RS [currp] page
- Update rear, front, currp.

- Count the page fault.
- Repeat steps 1 to 6 till reference string not over.
- Stop.

1. **LRU (Least Recently Used)-**

   It selects least recently used page of the main memory as a victim page. It refers to the past reference of the pages. The one which is not used for longest time is selected for replacement.

   In short LRU selects the farthest page in the left hand side direction of currently faulted page.

   LRU is implemented using **(a) counter (b) stack**.

(A)Using Counter

Data Structure-

1. M: total no of reference string.
2. RS: Stores reference string i.e. demand page number.
3. F: available n free frames.
4. Counter : is attached with each page . It counts page is referenced or not .Select smallest count page which is least recently used is selected for the victim page.

Algorithm

- Find out first free frame
- If free frame not available then free least recently used page from frame using past reference i.e. from RS[0] to RS[currp]

- Load RS[currp] into the LRU page frame
- currp++
- count the page fault
- Repeat step 1 to 5 till reference string is not over
- Stop.

(A)Using stack

Data Structure-

1. M: total no of reference string.
2. RS: Stores reference strings i.e. demand page number.
3. F: available n free frames.
4. Stack: To keep a stack of page numbers. Whenever a page is referenced, it is removed from stack and put on the top. In this way least recently used pages are found at bottom of the stack.

Algorithm

- Find out first free frame
- If free frame not available then free least recently used page stack (access the page which is stored at bottom)
- Load RS[currp] into the LRU page frame
- currp++
- count the page fault
- Repeat step 1 to 5 till reference string is not over
- Stop.

2. Second Chance Algorithm-

Data Structure-

1. M: total no of reference string.
2. RS: Stores reference strings i.e. demand page number.
3. F: available n free frames.
4. Ref_bit: Reference bit of page which stores 1 or 0.

Algorithm:

- Find out first free frame.
- If free frame is not available then

Ref_bit of page is checked serially, if it is 0 then the bit is set to 1and

page is replaced.

If it is 1 the bit is set to zero.

The first page along with reference bit 0 will appear at first position

and will be selected for replacement.

- Load F =RS [currp] page
- Count the page fault.
- Repeat steps 1 to 4 till reference string not over.
- Stop.

3. MFU(Most Frequently Used)-

Data Structure-

1. M: total no of reference string.
2. RS: Stores reference string I. E demand page number.
3. F: available n free frames.
4. Counter: A reference counter.

Algorithm:

- Find out first free frame.
- If free frame is not available then select MFU page using reference counter
- Load F =RS [currp] page
- Update currp and reference counter .
- Count the page fault.
- Repeat steps 1 to 5 till reference string not over.
- Stop.

4. **Optimal replacement-**

Data Structure-

1. M: total no of reference string.
2. RS: Stores reference string i.e. demand page number.
3. F: available n free frames.

Algorithm

- Find out first free frame
- If free frame not available then free optimal page from frame using

future reference i.e. from RS[currp] to RS[M]

- Load RS[currp] into the optimal page frame
- count the page fault
- Increment currp by one
- Repeat step 1 to 5 till reference string is not over
- Stop.

Note: Students can use suitable data structure , if necessary.

Set A:

Write the simulation program for demand paging and show the page scheduling and total number of page faults using following algorithms.Assume memory of n frames.Consider following page reference string :

a) 9,14,10,11,15,9,11,9,15,10,9,15,10,12,15

b) 1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6

1) Implementation of FIFO
2) Implementation of LRU using STACK
3) Implementation of LRU using COUNTER

Set B:

Write the simulation program for demand paging and show the page scheduling and total number of page faults using following

algorithms.Assume memory of n frames. Consider following page reference string :

a) 9,14,10,11,15,9,11,9,15,10,9,15,10,12,15
b) 1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6

1) Implementation of second chance.
2) Implementation of MFU

Set C:

Write the simulation program for demand paging and show the page scheduling and total number of page faults using following algorithms.Assume memory of n frames.Consider following page reference string :

a) 9,14,10,11,15,9,11,9,15,10,9,15,10,12,15
b) 1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6

1) Implementation of Optimal page replacement
2) Implementation of LFU
3) Implementation of MRU

Assignment Evaluation

0:Not Done [ ]   1:Incomplete [ ]   2.Late Complete [ ]
3:Needs Improvement [ ]   4:Complete [ ]   5:WellDone [ ]

Signature of the Instructor   Date of Completion

Assignment no : 5

Topic: FILE ALLOCATION METHODS

The main idea behind allocation is effective utilization of file space and fast access of the files.
There are three types of allocation:

1. Sequential (contiguous)
    - Each file occupies a set of contiguous blocks on the disk.
    - Simple – only starting location (block #) and length (number of blocks) are required.
    - Random access.
    - Wasteful of space (dynamic storage-allocation problem).
    - Files cannot grow.
2. Linked
    - Simple – need only starting address
    - Free-space management system – no waste of space
    - No random access
3. Indexed
    - Need index table
    - Random access
    - Dynamic access without external fragmentation, but have overhead of index block.

Set-A

Write a C Program to simulate Sequential (contiguous) File Allocation Method. Assume Disk of size‘d’, value of which should be taken from user. Use separate Tables (implemented using linked lists) to keep track of Used and Free space respectively. Make use of the following Menu to perform Operations:

- Allocate space for newly created file.
- Deallocate space for now-deleted file.
- Show Used and/or Free Space on Disk.
- Exit

Set-B

Write a C Program to simulate Linked File Allocation Method. Assume Disk of size 'd', value of which should be taken from user. Use separate Tables (implemented using linked lists) to keep track of Used and Free space respectively. Make use of the following Menu to perform Operations:

- Allocate space for newly created file.
- Deallocate space for now-deleted file.
- Show Used and/or Free Space on Disk.
- Exit

Set-C

Write a C Program to simulate Indexed File Allocation Method. Assume Disk of size'd', value of which should be taken from user. Use separate Tables (implemented using linked lists) to keep track of Used and Free space respectively. Make use of the following Menu to perform Operations:

- Allocate space for newly created file.
- Deallocate space for now-deleted file.
- Show Used and/or Free Space on Disk.
- Exit

Assignment Evaluation

0:Not Done [ ]   1:Incomplete [ ]   2.Late Complete [ ]
3:Needs Improvement [ ]   4:Complete [ ]   5:WellDone [ ]

Signature of the Instructor   Date of Completion