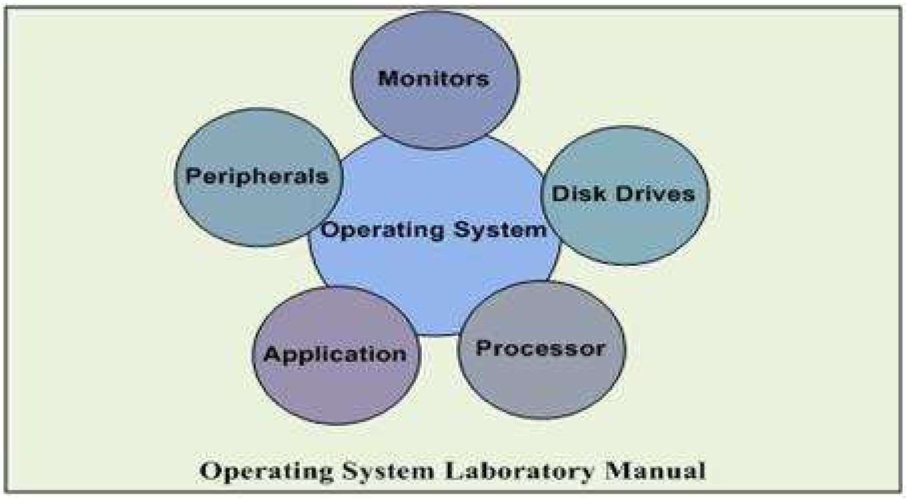
MUFFAKHAM JAH COLLEGE OF ENGINEERING AND TECHNOLOGY

Banjara Hills, Hyderabad, Telangana

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Operating System Laboratory Manual



Academic Year 2016-2017

Table of Contents

Part I

Contents

1. Vision of the Institution

To be part of universal human quest for development and progress by contributing high calibre, ethical and socially responsible engineers who meet the global challenge of building modern society in harmony with nature.

2. Mission of the Institution

- To attain excellence in imparting technical education from undergraduate through doctorate levels by adopting coherent and judiciously coordinated curricular and co-curricular programs.
- To foster partnership with industry and government agencies through collaborative research and consultancy.
- To nurture and strengthen auxiliary soft skills for overall development and improved employability in a multi-cultural work space.
- To develop scientific temper and spirit of enquiry in order to harness the latent innovative talents.
- To develop constructive attitude in students towards the task of nation building and empower them to become future leaders
- To nourish the entrepreneurial instincts of the students and hone their business acumen.
- To involve the students and the faculty in solving local community problems through economical and sustainable solutions.

3. Department Vision

To contribute competent computer science professionals to the global talent pool to meet the constantly evolving societal needs.

4. Department Mission

Mentoring students towards a successful professional career in a global environment through quality education and soft skills in order to meet the evolving societal needs.

5. Programme Education Objectives

1. Graduates will demonstrate technical skills and leadership in their chosen fields of employment by solving real time problems using current techniques and tools.

2. Graduates will communicate effectively as individuals or team members and be successful in the local and global cross cultural working environment.

3. Graduates will demonstrate lifelong learning through continuing education and professional development.

4. Graduates will be successful in providing viable and sustainable solutions within societal, professional, environmental and ethical contexts

6. Programme Outcomes

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Lifelong learning:** Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

7. Programme Specific Outcomes

The graduates will be able to:

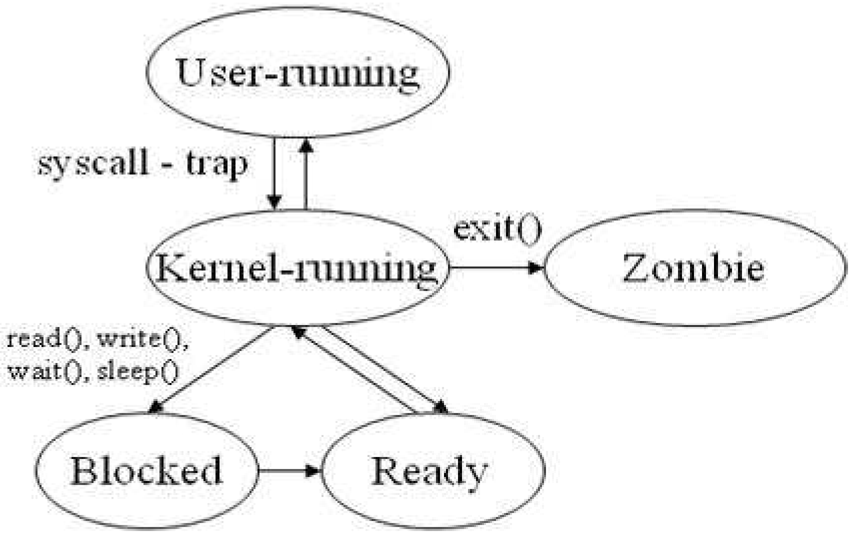
PSO1: Demonstrate understanding of the principles and working of the hardware and software aspects of computer systems.

PSO2: Use professional engineering practices, strategies and tactics for the development, operation and maintenance of software

PSO3: Provide effective and efficient real time solutions using acquired knowledge in various domains.

8. Introduction to Operating System Laboratory

Introduction to Operating System Laboratory Operating System:
Operating system is an interface between the user and the computer hardware.



Process System Calls

The fork() system call
To create a new process, we must use the fork() system call. A process calling fork() spawns a child process. fork() causes the UNIX system to create a new process, called the "child process", with a new process ID. The contents of the child process are identical to the contents of the parent process. The child is almost an identical clone of the parent:

- Program Text (segment .text)
- Stack (ss)
- PCB (eg. registers)
- Data (segment .data)

The fork() is one of the those system calls, which is called once, but returns twice! After fork() both the parent and the child are executing the same program. On error, fork() returns -1 . To differentiate which process is which, fork() returns zero in the child process and non-zero (the child's process ID) in the parent process. Remember, after fork() the execution order is not guaranteed.

The exec() system call

The exec() call replaces a current process' image with a new one (i.e. loads a new program within current process).

The new image is either regular executable binary file or a shell script. There's no a syscall under the name exec(). By exec() we usually refer to a family of calls:
− int execl(char *path, char *arg, ...);
– int execv(char *path, char *argv[]);
– int execle(char *path, char *arg, ..., char *envp[]);
– int execve(char *path, char *argv[], char *envp[]);
– int execlp(char *file, char *arg, ...);
– int execvp(char *file, char *argv[]);

Here's what l, v, e, and p mean:
– l means an argument list,
– v means an argument vector,
– e means an environment vector, and
– p means a search path

The wait() System Call

We can control the execution of child processes by calling wait() in the parent. wait() forces the parent to suspend execution until the child is finished. wait() returns the process ID of a child process that finished. If the child finishes before the parent gets around to calling wait(), then when wait() is called by the parent, it will return immediately with the child's process ID. (It is possible to have more than one child process by simply calling fork() more than once.).

The exit() system call:

The exit() system call ends a process and returns a value to it parent. The prototype for the exit() system call is:
void exit(status)
int status;
where status is an integer between 0 and 255. This number is returned to the parent via wait() as the exit status of the process. By convention, when a process exits with a status of zero that means it didn't encounter any problems; when a process exit with a non-zero status that means it did have problems.

The getpid() and getppid() system calls

Synopsis

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Description getpid() returns the process ID of the calling process. getppid() returns the process ID of the parent of the calling process.

INTERPROCESS COMMUNICATION

Inter-Process communication (IPC), is the mechanism whereby one process can communicate with another process, i.e exchange data. IPC in linux can be implemented using pipe, shared memory, message queue, semaphore, signal or sockets.

Pipe

Pipes are unidirectional byte streams which connect the standard output from one process into the standard input of another process. A pipe is created using the system call pipe that returns a pair of file descriptors. The descriptor pfd[0] is used for reading and pfd[1] is used for writing. Can be used only between parent and child processes.

Shared memory

Two or more processes share a single chunk of memory to communicate randomly. Semaphores are generally used to avoid race condition amongst processes. Fastest amongst all IPCs as it does not require any system call. It avoids copying data unnecessarily.

Message Queue A message queue is a linked list of messages stored within the kernel A message queue is identified by a unique identifier Every message has a positive long integer type field, a non-negative length, and the actual data bytes. The messages need not be fetched on FCFS basis. It could be based on type field.

Semaphores

A semaphore is a counter used to synchronize access to a shared data amongst multiple processes. To obtain a shared resource, the process should: o Test the semaphore that controls the resource. o If value is positive, it gains access and decrements value of semaphore. o If value is zero, the process goes to sleep and awakes when value is > 0. When a process relinquishes resource, it increments the value of semaphore by 1. Producer-Consumer problem A producer process produces information to be consumed by a consumer process A producer can produce one item while the consumer is consuming another one. With bounded-buffer size, consumer must wait if buffer is empty, whereas producer must wait if buffer is full. The buffer can be implemented using any IPC facility. Exp# 5a Fibonacci & Prime Number Aim To generate 25 fibonacci numbers and determine prime

An Introduction to the File System

INODES

In addition to it's actually data contents, every file in the FreeBSD file system has a unique descriptor called inode. An inode contains the information of the file which includes at least:

- file owner: the owner (normally the creator) of the file
- file type: regular, directory, etc
- access permissions: specifies who have the right to read, write, and execute the file
- file access times: when it was last modified, accessed, etc.

- file size: data in a file is stored in a virtual address starting from byte offset 0 to the highest offset byte. The size of a file is address of the highest byte.
- #of links: the number of names the file has.

Data stored in an inode may be accessed either through the ls command or the stat system call, which will be discussed shortly.

Structure of a Regular File
The data in file is saved as a logical stream of bytes, although the actually physical storage may be over multiple sectors and tracks. This logical stream of bytes starts from offset zero to the last byte with the highest offset. One can read data of variable length (limited by the actual length of the file) starting from any offset, and one can write data of any length from any offset, even an offset that is beyond the last byte of the file. For example, one can write 10 bytes of data starting from offset 1000 to a file with only one byte in it. After the write is completed, The size of the file becomes 1009, with a gap between the first bytes and the last ten bytes just written by the write. When a read tries to read data from inside the gap, the kernel treats the gap as if it were filled with zeros.

File Access Permissions
The FreeBSD file system protects files according to three classes: the owner and the group owner of the file, and other users. Each classes may be given the right to read, write, and execute the file. the three access rights are specified by three binary digits in the order of read (r), write (w) and execute (x).

System calls for File Processing
FreeBSD (4.4) has six file-related system calls. The following table briefly describe the function of each.

System calls	**Function**
open	open an existing file or create a new file
read	Read data from a file
write	Write data to a file
lseek	Move the read/write pointer to the specified location
close	Close an open file
unlink	Delete a file
chmod	Change the file protection attributes
stat	Read file information from inodes

Files to be included for file-related system calls.

```
#include        <unistd.h>
#include        <fcntl.h>
#include        <sys/types.h>
#include        <sys/uio.h>
#include        <sys/stat.h>
```

Open files

The open system call can be used to open an existing file or to create a new file if it does not exist already. The syntax of open has two forms:

*int open(const char *path, int flags);* and
*int open(const char *path, int flags, mode_t modes)*;

The first form is normally used to open an existing file, and the second form to open a file and to create a file if it does not exist already. Both forms returns an integer called the file descriptor. The file descriptor will be used for reading from and writing to the file. If the file cannot be opened or created, it returns -1. The first parameter path in both forms sPecifies the file name to be opened or created. The second parameter (flags) specifies how the file may be used. The following list some commonly used flag values.

Flag	**Description**
O_RDONLY	open for reading only
O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_NONBLOCK	do not block on open
O_APPEND	append on each write
O_CREAT	create file if it does not exist
O_TRUNC	truncate size to 0
O_EXCL	error if create and file exists
O_SHLOCK	atomically obtain a shared lock
O_EXLOCK	atomically obtain an exclusive lock
O_DIRECT	eliminate or reduce cache effects
O_FSYNC	synchronous writes
O_NOFOLLOW	do not follow symlinks

The flag (*O_CREAT*) may be used to create the file if it does not exist. When this flag is used, the third parameter (*modes*) must be used to specify the file access permissions for the new file. Commonly used modes (or access permissions) include

Constant Name	**Octal Value**	**Description**
S_IRWXU	0000700	/* RWX mask for owner */
S_IRUSR	0000400	/* R for owner */
S_IWUSR	0000200	/* W for owner */
S_IXUSR	0000100	/* X for owner */
S_IRWXO	0000007	/* RWX mask for other */
S_IROTH	0000004	/* R for other */
S_IWOTH	0000002	/* W for other */
S_IXOTH	0000001	/* X for other */

R: read, W: write, and X: executable

For example, to open file "tmp.txt" in the current working directory for reading and writing:
fd = open("tmp.txt", O_RDWR);

To open "sample.txt" in the current working directory for appending or create it, if it does not exist, with read, write and execute permissions for owner only:
fd = open("tmp.txt", O_WRONLY |O_APPEND |O_CREAT, S_IRWXU);

A file may be opened or created outside the current working directory. In this case, an absolute path and relative path may prefix the file name. For example, to create a file in /tmp directory:
open("/tmp/tmp.txt", O_RDWR);

Read from files
The system call for reading from a file is ***read.*** Its syntax is
*s*size_t read(int fd, void *buf, size_t nbytes);

The first parameter *fd* is the file descriptor of the file you want to read from, it is normally returned from open. The second parameter *buf* is a pointer pointing the memory location where the input data should be stored. The last parameter *nbytes* specifies the maximum number of bytes you want to read. The system call returns the number of bytes it actually read, and normally this number is either smaller or equal to *nbytes.* The following segment of code reads up to 1024 bytes from file tmp.txt:

```
int actual_count = 0;
int fd = open("tmp.txt", O_RDONLY);
  void *buf = (char*) malloc(1024);

actual_count = read(fd, buf, 1024);
```

Each file has a pointer, normally called read/write offset, indicating where next *read* will start from. This pointer is incremented by the number of bytes actually read by the *read* call. For the above example, if the offset was zero before the *read* and it actually read 1024 bytes, the offset will be 1024 when the *read* returns. This offset may be changed by the system call *lseek*, which will be covered shortly.

Write to files
The system call *write* is to write data to a file. Its syntax is
*s*size_t write(int fd, const void *buf, size_t nbytes);
It writes *nbytes* of data to the file referenced by file descriptor *fd* from the buffer pointed by*buf.* The *write* starts at the position pointed by the offset of the file. Upon returning from *write*, the offset is advanced by the number of bytes which were successfully written. The function returns the number of bytes that were actually written, or it returns the value -1 if failed.

Reposition the R/W offset
The *lseek* system call allows random access to a file by reposition the offset for next read or write. The syntax of the system call is
off_t lseek(int fd, off_t offset, int reference);
It repositions the offset of the file descriptor *fd* to the argument *offset* according to the directive reference. The *reference* indicate whether *offset* should be considered from the beginning of the file (with *reference* 0), from the current position of the *read/write* offset (with *reference*1), or from the end of the file (with *reference* 2). The call returns the byte offset where the next *read/write* will start.

Close files
The *close* system call closes a file. Its syntax is
int close(int fd);
It returns the value 0 if successful; otherwise the value -1 is returned.

Delete files
The *unlink* may be used to delete a file (A file may have multiple names (also called links), here we assume that a file in this context has only one name or link.). Its syntax is:
*int unlink(const char *path);*
path is the file name to be deleted. The *unlink* system call returns the value 0 if successful, otherwise it returns the value -1.

Change file access permissions

File access permissions may be set using the *chmod* system call (note that there is a command with the same name for setting access permissions.). It has two forms:
*int chmod(const char *path, mode_t mode); or int fchmod(int fd, mode_t mode);*

Both forms set the access permission of a file to mode. In the first form the file is identified by its name, and in the second it is identified by a file descriptor returned from the *open* system call. For *mode*, you may use any of the constants defined in the *Open files* section of this tutorial.
The system call returns the value 0 if successful, otherwise it returns the value -1.

Accessing file information from Inodes The stat system call can be used to access file information of a file from its inode. It can appear in two forms:

*int stat(const char *path, struct stat *sb); or int fstat(int fd, struct stat *sb);*

Both forms return the information through the stat structure pointed by sb. In the first form, the file is identified by its name and in the second form, it is identified by its file descriptor returned from a call to open. The stat structure includes at least the following elements:

Element	**Description**
st_mode	file protection mode
st_uid	User ID of the file owner
st_size	file size in bytes

No permission is needed to *stat* a file. However since the second form requires a file descriptor of the file and a file descriptor may be only obtained by *open, fstat* can only be applied to files that has proper access permissions.
The call returns the value 0 if successful, otherwise it returns the value -1. The call fails if the specified path or the file does not exist.

File Locking Using fcntl() call

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fildes, int cmd, ...);
struct flock {
      short l_type;
      short l_whence;
      off_t l_start;
      off_t l_len;
      pid_t l_pid;
};
struct flock64 {
      short l_type;
      short l_whence;
      off64_t l_start;
      off64_t l_len;
      pid_t l_pid;
};
```

The fcntl() function provides control of open file descriptors.
The following commands are supported for all file types:

F_DUPFD

Return a new file descriptor which is the lowest numbered available (that is, not already open) file descriptor greater than or equal to the specified argument, which is of type int The new file descriptor refers to the same open file description as the original file descriptor, and shares any locks. The FD_CLOEXEC flag associated with the new file descriptor is cleared to keep the file open across calls to one of the exec() family of functions. The return value is the new file descriptor on success, or -1 on error.

F_SETFD

Set the file descriptor flags for the specified file descriptor. The argument is the new set of flags, as a variable of type int. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file. The return value is 0 on success, or -1 on error. The following file descriptor flags may be set. Any additional bits set from the flags specified for F_GETFD are ignored. If any bits not defined here are specified, behavior is undefined.

FD_CLOEXEC

If set, the file descriptor is closed when one of the exec() family of functions is called. If not set, the file descriptor is inherited by the new process image.

F_GETFD

Get the file descriptor flags for the specified file descriptor. This command takes no argument. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file. The return value is the current file descriptor flags on success, or -1 on error. In addition to the flags specified for F_SETFD, the following flags may be returned.

FD_MANDATORYLOCK
Mandatory locking is enabled for the file referred to by the specified file descriptor.

FD_ADVISORYLOCK
Advisory locking is enabled for the file referred to by the specified file descriptor.

FD_DIRECTORY
The specified file descriptor refers to a directory.

F_SETFL

Set the file status flags for the specified file descriptor. The argument is the new set of flags, as a variable of type int. These flags are as specified for the *oflag* argument to open(), along with the additional values specified later. Bits corresponding to the file access mode and any other *oflag* bits not listed here are ignored. If any bits not defined here or in open() are set, behavior is undefined. The return value is 0 on success, or -1 on error. The following file status flags can be changed with F_SETFL:

SETFL:

O_APPEND

Valid only for file descriptors that refer to regular files. The file pointer is moved to the end of the file before each write.

O_ASYNC

Valid only for file descriptors that refer to sockets and communications ports. If enabled for a file descriptor, and an owning process/process group has been specified with the F_SETOWN command to fcntl(), then a SIGIO signal is sent to the owning process/process group when input is available on the file descriptor.

O_BINARY
Sets the file descriptor to binary mode. See PORTING ISSUES for more information.

O_LARGEFILE
Sets the file descriptor to indicate a largefile aware application.

O_NDELAY
Sets the file descriptor to no-delay mode.

O_NONBLOCK Sets the file descriptor to non-blocking mode. The distinction between non-blocking mode and no-delay mode is relevant only for a few types of special files such as pipes and FIFOs. Refer to read() and write() for more information.

O_SYNC
Sets the file descriptor to synchronous-write mode. Writes do not return until file buffers have been flushed to disk.

O_TEXT
Sets the file descriptor to text mode. See PORTING ISSUES for more information.

FAPPEND
A synonym for O_APPEND.

FASYNC
A synonym for O_ASYNC.

FNDELAY
A synonym for O_NDELAY.

F_GETFL

Get the file status flags and file access modes for the specified file descriptor. These flags are as specified for the *oflag* argument to open(), along with the additional values described for F_SETFL. File status flags and file access modes are associated with the file description and do not affect other file descriptors that refer to the same file with different open file descriptions. The return value is the current file status flags and file access modes on success, or -1 on error. The following macros can be used to access fields of the return value:

O_ACCMODE
Extracts the access-mode field, which is one of O_RDONLY, O_RDWR, or O_WRONLY. Refer to the documentation for open() for more information. The following commands are supported only for sockets and communication ports:

F_SETOWN
Sets the owning process ID or process group ID for the specified file descriptor. The

owning process or process group can receive SIGURG signals for out-of-band data or sockets and/or SIGIO signals for readable data on sockets or communications ports. The argument is the process ID or the negative of the process group ID for the owner, as a variable of type pid_t. The return value is 0 on success, or -1 on error.
To receive SIGURG signals, the process should establish a SIGURG handler prior to setting ownership of the file descriptor. A SIGURG signal is generated whenever outof-band data is received. To receive SIGIO signals, the process should establish a SIGIO handler prior to setting ownership of the file descriptor, and then must enable O_ASYNC with the F_SETFL command to fcntl(). A SIGIO signal is generated whenever there is data to be read on the file descriptor.

F_GETOWN Gets the owning process ID or process group ID for the specified file descriptor. The return value is the owner ID on success, or -1 on error. Behavior is undefined if no owner has been established with F_

SETOWN The following commands are used for file locking. Locks may be advisory or mandatory; refer to PORTING ISSUES for more information. These command are supported only for regular files:

F_GETLK
Get the first lock which blocks a lock description for the file to which the specified file descriptor refers. The argument is a pointer to a variable of type struct flock, described later. The structure is overwritten with the returned lock information. If no lock is found that would prevent this lock from being created, then the structure is unchanged except for the lock type, which is set to F_UNLCK. The return value is 0 on success, or -1 or error.

F_GETLK64
Equivalent to F_GETLK, but takes a struct flock64 argument rather than a struct flock argument.

F_SETLK
Set or clear a file segment lock for the file to which the specified file descriptor refers. The argument is a pointer to a variable of type struct flock, described later. F_SETLK is used to establish shared (or read) locks (F_RDLCK) or exclusive (or write) locks (F_WRLCK), as well as to remove either type of lock (F_UNLCK). The return value is 0 on success, or -1 on error. If the lock cannot be immediately obtained, fcntl() returns -1 with errno set to EACCES.

F_SETLK64
Equivalent to F_SETLK, but takes a struct flock64 argument rather than a struct flock argument.

F_SETLKW
This command is the same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the thread waits until the request can be satisfied. If a signal that is to

be caught is received while fcntl() is waiting for a region, fcntl() is interrupted. Upon return from the signal handler, fcntl() returns -1 with errno set to EINTR, and the lock operation is not done.

F_SETLKW64
Equivalent to F_SETLKW, but takes a struct flock64 argument rather than a struct flock argument.
When a shared lock is set on a segment of a file, other processes can set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock fails if the file descriptor was not opened with read access.
An exclusive lock prevents any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock fails if the file descriptor is not opened with write access. The flock and flock64 structure contains the following fields:

l_type
Specifies the type of lock request. Valid settings are:

F_RDLCK
Create a shared lock.

F_WRLCK
Create an exclusive lock.

F_UNLCK
Remove a lock.

l_whence
Specifies the starting offset of the lock segment in the file. Valid settings are:

SEEK_SET
l_start specifies a position relative to the start of the file.

SEEK_CUR
l_start specifies a position relative to the current file offset.

SEEK_END
l_start specifies a position relative to the end of the file.
On a successful return from an F_GETLK or F_GETLK64 command for which a lock was found, the l_whence field is set to SEEK_SET.

l_start
Specifies the relative offset of the start of the lock segment. This setting is used with l_whence to determine the actual start position.

l_len
Specifies the number of consecutive bytes in the lock segment. This value may be negative.

l_pid
On a successful return from an F_GETLK or F_GETLK64 command for which a lock was found, this field contains the process ID of the process holding the lock.

If l_len is positive, the affected area starts at l_start and ends at (l_start + l_len - 1). If l_len is negative, the area affected starts at (l_start + l_len), and ends at (l_start - 1). Locks may start and end beyond the current end of a file, but must not be negative relative to the beginning of the file. Setting l_len to 0 sets a lock that can extend to the largest possible value of the file offset for that file. If such a lock also has l_start set to 0 and l_whence set to SEEK_SET, the whole file is locked.

There can be at most one type of lock set of each byte in the file. Before a successful return from an F_SETLK, F_SETLK64, F_SETLKW, or F_SETLKW64 request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region is replaced by the new lock type. As specified earlier, an F_SETLK, F_SETLK64, F_SETLKW, or F_SETLKW64 request fails or blocks, respectively, when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

All locks associated with a file for a given process when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process created using fork().

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock another process' locked region. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, fcntl() returns -1 with errno set to EDEADLK.

System Requirement

Systems/Software Requirements: Intel based desktop PC with minimum of 166 MHZ or faster processor with at least 64 MB RAM and 100 MB free disk space Turbo C or TC3 complier in Windows XP or Linux Operating System.

Part II

Programs

Program 1

To demonstrate the usage of fork, wait, getpid and getppid system calls

Problem Definition

To demonstrate the usage of fork, wait, getpid and getppid system calls

Problem Description

The program is used to display the ID's of the processes and its parents. The fork function creates a copy of the running process. The copy (the child) has a copy of parent process stack, data area, heap and starts after fork statement. Now two processes are running simultaneously one is parent process and second is child process. The child process will be in the execution state when PID = 0 and PID is greater than 0 parent process will be in the execution state. The wait function suspends the execution of current process until a child has exited or until a signal is delivered whose action is to terminate the current process. The sleep() suspends the execution of a process for a specified amount of time. The getpid function returns the ID of the process from where it is called and getppid function returns the ID of the parent's process from where it is called.

Algorithm

Step 1: Process ID of program and its parent is fetched and displayed.

Step 2: A child process is created using fork() and its pid is obtained.

Step 3: If child process creation was not successful, then the function exits displaying error.

Step 4: In the child process, the process id of child, its parent and the parent's parent is displayed and child is made to sleep for 3 seconds.

Step 5: The child is terminated and the parent prints some messages and the program ends.

Problem Validation

Input:

To get process ID and parent ID of processes execute the steps of the above algorithm.

Output:

In child process:

ID of the child process is 8599

ID of child's parent is 8598

In parent process:

ID of the parent process is 8599

ID of the parent's parent is 9952

Program 2

To show the existence of an orphan process.

Problem Definition

To show the existence of an orphan process.

Problem Description

An orphan process is a computer process whose parent process is dead. A process can become an orphan during remote invocation when the client process crashes after making a request of the server. A process can also be orphaned during run time on the same machine as its parent process. In UNIX-like operating system, any orphaned process will be immediately adopted by the special “init” system process. This operation is called re-parenting and occurs automatically. Even though, technically, the process has the “init” process as its parent, it is still called an orphan process since the process which originally created it, no longer exists.

Algorithm

Step 1: In a parent process, the child process is created using fork ().

Step 2: The child process is put to sleep while the child is still executing.

Step 3: The parent process execution completes and it goes to child to execute.

Step 4: Now the child became orphan and displays its parent process ID and child ID.

Step 5: The program terminates.

Problem Validation

Input:

Execute the steps of the algorithm in the given order.

Output:

In parent process:

IParent ID is 1201

Parent’s parent ID is 1103

In child process:

Child ID is 1202

Child’s parent ID is 1.

Program 3

To show the existence of a zombie process

Problem Definition

To show the existence of a zombie process

Problem Description

In UNIX operating system, a zombie process is a process that has completed the execution but still has an entry in the process table, allowing the process that started to read its exit status. In other words, the child process has died but has not been reaped. When a process ends, all of the memory and resources associated with it are deallocated so that they can be used by other processes. However, the processes entry in the process table remains. The parent is sent a signal indicating that the child has died; the handler for this signal will typically execute the system call, which reads the exit state and removes the zombie.

Algorithm

Step 1: A child process is created using fork ().

Step 2: The process id of child, parent and grandparent is displayed and the child process is put to sleep for a longtime.

Step 3: The program is terminated and the process table is checked for a zombie process by the command "ps - l"

Step 4: In the process table it displays the state of child process as Zombie.

Problem Validation

Input:

Execute the steps of the algorithm in the given order.

Output:

Child ID is 1202

Child's parent ID is 1201

Parent ID is 1201

Parent's parent ID is 1103

In the process table it displays the process state as Zombie.

Program 4

To execute programs using execl, execlp, execv, execvp system calls

Problem Definition

To execute programs using execl, execlp, execv, execvp system calls

Problem Description

The execl function initiates a new program in the same environment in which it is operating. An executable program with fully qualified path, i.e, /bin/ls and arguments are passed to the function. The execlp routine will perform the same as excel that it will use environment variable path to determine which is executable to process. Thus, a fully qualified name would not have to be used. The first argument to the function could be "ls". This function can also take the fully qualified name as it also resolves explicitly. The execv This is same as exec except that the arguments are passed as null terminated array of pointers to char. The first element "argv[0]" is the command name. The execvp routine will perform the same purpose except that it will use environment variable path to determine which is executable to process. The first argument to the function could be "ls". This function can also take the fully qualified name as it also resolves explicitly.

Algorithm

Step 1: Two user defined programs are created or specify the programs using /bin

Step 2: The functions of exec () family are conditioned using switch statement.

Step 3: During execution, the user selects a function procedure.

Step 4: The exec () family uses previously created user defined programs to give the output.

Problem Validation

Input:

Execute the steps of the algorithm in the given order.

Output:

Child ID is 1202

Child's parent ID is 1201

Parent ID is 1201

Parent's parent ID is 1103

In the process table it displays the process state as Zombie.

Program 5

To demonstrate the file related system calls

Problem Definition To demonstrate the file related system calls

Problem Description

A system call is just what its name implies, i.e, a result for the operating system to do something on behalf of users program. The system calls are functions used in Kernel itself.

File Related System Calls:

open () —> For opening an existing file or creating a new file.

read () —> For reading data from a file.

write () —> For writing data to a file.

close () —> For closing an open file.

Commonly Used Flags:

O_RDONLY —> open for reading only.

O_WRONLY —> open for writing only.

O_RDWR —> open for reading and writing.

O_APPEND —> append on each write.

O_CREAT —> create a file if it does not exist. The program copies the contents of the source file and places the same in to the destination file.

Algorithm

Step 1: File descriptor of two files are created using open ().

Step 2: File descriptors are checked for errors and program is exited if any error occurs.

Step 3:If no errors occur, then the contents of first file are written into second file via a buffer using read () and write () system calls.

Step 4: Step 3 is repeated until EOF is encountered and the file descriptors are closed ending the program.

Problem Validation

Input:

Input file and output file are passed as command line arguments.

Output:

Copy's the contents from input file to output file.

Program 6

To demonstrate file locking using 'fcntl' system call.

Problem Definition To demonstrate file locking using 'fcntl' system call.

Problem Description

File locking is a mechanism that restricts access to a computer file by only allowing one user (or) process access at any specific time.

The two most common mechanisms for file locking in UNIX are: fcntl and flock. The functions of fcntl are:

F_GETLK: Get the first lock which blocks the lock description pointed to by the third argument.

F_SETLK: Set or clear a file segment lock according to the lock description pointed by the 3rd argument taken as a pointer to type struct flock defined in ¡fcntl.h¿

F_SETLKW: This command shall be equivalent to F_SETLK except that if a shared or exclusive lock is blocked by other locks, the thread shall wait until the request can be satisfied.

F_RDLCK: This macro is used to specify a read lock.

F_WRLCK: This macro is used to specify a write lock.

F_UNLCK: This macro is used to specify that the region is unlocked.

Algorithm

Step 1: lk and uk of type struct flock are created.

Step 2: All the lk, uk are initialized to 0.

Step 3:By using the fcntl, we are going to implement F_SETLKW.

Step 4: The fcntl is set for both the lk and uk.

Problem Validation

Input:

Input file and output file are passed as command line arguments.

Output:

Copy's the contents from input file to output file.

Program 7

To print the attributes of a file.

Problem Definition To print the attributes of a file.

Problem Description

File attribute is a metadata which describes (or) is associated with a computer file. The operating system maintains certain properties associated with every file and directories in addition to

file contents. Such properties include:

File size in bytes

File date/time (creation time, access time, etc)

Mode number of file

Inode number of file

Modification time, etc

These properties are maintained by various components of the operating system and they are handled automatically.

Algorithm

Step 1: A stat structure variable is created.

Step 2: The variable is pointed to the file via command-line argument.

Step 3: The variable is associated with stat keywords to obtain the attributes of the file and the appropriate message is printed.

Step 4: The program ends.

Problem Validation

Input:

Input file is passed as command line arguments.

Output:

```
File size: 1330 bytes

Block allocated:

Last status change: wed Sep 16 15:21:33 2015

Last file access: wed Sep 16 15:22:29 2015

Last file modification: wed Sep 16 15:22:29 2015
```

Program 8

To print the file type for each command line

Problem Definition To print the file type for each command line

Problem Description

The program displays the file type of each input file. There are different types of files like regular, directory, device, block, skit, symbolic, etc can be identified using flags given below.

S_ISDIR: It checks the file mode to see whether the file is in directory or not. If so, it returns true.

S_ISLNK: It checks the file mode to see whether the file is a symbolic link or not. If so, it returns true.

S_ISFIFO: It checks the file mode to see whether a file is FIFO (a named pipe) or not.
S_ISREG: It checks the file mode to see whether a file is regular or not.

S_ISCHR: It checks the file mode to see whether a file is character device file or not.

S_ISSOCK: It checks the file mode to see whether the file is a socket or not.

Algorithm

Step 1 : A stat structure variable is created.

Step 2 : The variable is pointed to the file via command-line argument.

Step 3 : Switch construct is used to test for the type of file.

Step 4 : Appropriate message is printed after testing is completed.

Step 5 : The program ends.

Problem Validation

Input:

N number of input files is passed as command line arguments.

Output:

Prints the file type for each command line argument.

Ex: f1.txt – regular file

D1 – directory file

Program 9

To display the file contents of a directory.

Problem Definition To display the file contents of a directory.

Problem Description

The program is used to display the contents of a directory including sub directories and files in the directory. To open the directory we can use opendir() and if directory exists reads all the files in that directory and display . If no directory exists with that name them return null.

Algorithm

```
Step 1 : Get directory name as command line argument.
Step 2 : If directory does not exist then stop.
Step 3 : Open the directory using opendir system call that returns a structure
Step 4 : Read the directory using readdir system call that returns a structure
Step 5 : Display d_name member for each entry.
Step 6 : Close the directory using closedir system call.
Step 7 : Stop
```

Problem Validation

Input:

Directory name is passed as command line arguments.

Output:

The files and subdirectories in the directory was listed that includes hidden files.

```
a.out ..

stat.c

dirlist.c

fork.c

exec.c
```

Program 10

To demonstrate basic operations with pipes.

Problem Definition To demonstrate basic operations with pipes.

Problem Description

To create a simple pipe with ‘C’, we make use of the pipe() system call. It takes a single argument which is an array of two integers, and if successful, the array will contain two new file descriptors to be used for the pipeline. After creating a pipe, the process typically spawns a new process. (Also, the child inherits the open file descriptors)

System Call: pipe()

Prototype: int pipe (int fd[2]);

Return: 0 on success

-1 on errors

fd[0] is set up for reading and fd[1] is set up for writing. The first element in the array (element 0) is set up and opened for reading while the second element (element 1) is setup and opened for writing. Visually speaking, the output of d1 becomes the input for fd(). Once again, all the data travelling through pipe moves through the kernel.

Algorithm

Step 1 : A pipe and a process is created.

Step 2 : If the process is a child process, we write into the pipe using write ().

Step 3 : If the process is a parent process, we read from the pipe using the buffer ‘buff’ of size using read ().

Step 4 : The program terminates.

Problem Validation

Input:

Execute the steps of the algorithm in the given order.

Output:

```
Hai
```

Program 11

To demonstrate echo server using pipes.

Problem Definition To demonstrate echo server using pipes.

Problem Description

Pipe acts as a medium to transfer output of one process as input to another process. The process will do the work and generate output. If the same output is required by the second process, then we can repeat the process. But it is time consuming. So we prefer to give output of first to second through the Inter Process Communication technique.

Echo Server: Any program that provides some service or work.

Algorithm

Step 1 : Two pipes a and b are created using pipe().

Step 2 : The child process/client writes into the pipe and sleeps for 5 seconds.

Step 3 : When the child process/client sleeps, the control is transferred to the parent process/server.

Step 4 : The server reads from the pipe and prints what the client has to say.

Step 5 : The server then, writes (gives response) into another pipe and goes into wait status.

Step 6 : The control is again transferred to the client where it reads from the pipe and prints what the server has to say.

Step 7 : The program terminates.

Problem Validation

Input:

Execute the steps of the algorithm in the given order.

Output:

In parent process- Hai from child

In child process- Hello from parent

Program 12

To demonstrate basic operations with message queues

Problem Definition

To demonstrate basic operations with message queues

Problem Description

In message queue, a mail box is created. The OS should read and write program. A message queue is created and owned by one process. A message queue is created by a msgget call with the following syntax:

msgget(key, flag)

The OS maintains an array of message queues and their keys. The first msgget call results in creation of new message queue. The position of the message queue in the system array (called the message queue id) is returned by the msgget call. This id is used in a send or receive call. The send and receive calls have the following syntax:

msgsnd(msgqid, msg, count, flag);

msgrcv(msgqid, msg, struct-ptr, maxcount, type, flag);

The count and flag parameters of msgsnd specify number of bytes n a message and the actions as should take if sufficient space is not available in message queue (e.g. Block the sender). In msgrcv call, msg-struct-ptr is the address of the structure to receive the message, maxcount is maximum length of message, type indicates the type of message to be received. When a message is sent to a message queue on which many processes are waiting, the operating system wakes all processes. When type parameter in msgrcv is positive, the operating system returns the first message with a matching type and if it is negative, the operating system returns the lowered number message with the type value less than the absolute value specified in the call.

Algorithm

Step 1 : A structure named mbuff is created having parameters an array 'a[10]'of type char and 'type' of type int.

Step 2 : A unique key is generated using msgget() and msqid .

Step 3 : A process is created, if it is a child process, a string is written and is assigned to the structure mbuff . The message is sent using the msgsnd() function.

Step 4 : The parent process receives the message from the child process using the msgrcv() and reads the message using the buffer. It prints the message received by the child process.

Step 5 : The program terminates.

Problem Validation

Input:

Execute the steps of the algorithm in the given order.

Output:

Hello

Program 13

To demonstrate echo server using message queues

Problem Definition

To demonstrate echo server using message queues

Problem Description

Echo server is any program that provides some service or work. In message queues, a mail box is created. OS should read and write program. Context switches makes program longer and therefore, shared memory is used. In message queues, the syntax for creating and receiving is as follows:

```
Create:
int msgid = msgget( (key_t) 21, IPC_CREAT|0600);

Sending Message:
msgsnd( msqid, &mbuff, length, 0);
struct mbuff
{
char[10];
int type;
};
struct mbuff a1;
strcpy(a1.a, "CSE_OSLAB");
a1.type = 1;

Receiving Message:
msgrcv( msqid, &buffer, length, type, 0);
```

Algorithm

Step 1 : : A unique key is generated using msgget () and msgid and a process is created using fork()).

Step 2 : The child process/client send a message using msgsnd() and sleeps for 1 second.

Step 3 : When the child process/client sleeps, the control is transferred to the parent process/server.

Step 4 : The server reads the message and prints what the client has to say.

Step 5 : The server then, sends (gives response) a message using msgsnd() and goes into wait status.

Step 6 : The control is again transferred to the client where it receives (reads) the message using msgrcv() and prints what the server has to say.

Step 7 : The program terminates.

Problem Validation

Input:

Execute the steps of the algorithm in the given order.

Output:

```
1604-14-733-006
```

Pass

Program 14

To demonstrate basic operations with shared memory

Problem Definition

To demonstrate basic operations with shared memory

Problem Description

For communication between process P1 and process P2 and if they are in the same processor shared memory is required. In message queues, the context switches make program longer, so shared memory is used. Shared memory is faster than message queues to send and receive messages because the processes will be in the same system. To use shared memory we have to do the following:

- create the shared memory
- attach the shared memory
- use it i.e. read or write it

Create the shared memory:

1. int shmid;
   shmid = shmget((key_t) 21, 20, IPC_CREAT|0600);
2. Attach the shared memory:
   char p;
   p = shmat (shmid, 0, 0)
3. Use the Shared Memory: strcpy(p, "MJCET"); => writing printf("%s
   n", p); => reading

Algorithm

Step 1 : A unique key is generated using shmget () and shmid.

Step 2 : A process is created. The parent process creates the shared memory

Step 3 : The child process attaches to the shared memory and writes a message into it.

Step 4 : The parent process reads the message and prints it.

Step 5 : The program terminates.

Problem Validation

Input:

Execute the steps of the algorithm in the given order.

Output:

MJCET

Program 15

To demonstrate basic operations with Semaphores

Problem Definition

To demonstrate basic operations with Semaphores

Problem Description

Semaphores are used for synchronization. If multiple processes share a common resource, they need a way to be able to use that resource without disrupting each other. A semaphore enforces mutual exclusion and controls access to the process critical sections. Only one process at a time can call the function. It will also allow or disallow access to the resource depending on how it is set up. For example, a semaphore which allowed any number of processes to read from the resource but only one could ever be in the process of writing to that resource at a time. It has three key stages:

- Create
- Initialize
- use

Algorithm

Step 1 : A unique key is generated using semget() and semid .

Step 2 : Using semctl, we set all the values that are stored in the array s[3] and we get all the values and store them into the array s1[3].

Step 3 : We use a for conditional loop to print the values stored in the s1[3] array.

Step 4 : The program terminates.

Problem Validation

Input:

Execute the steps of the algorithm in the given order.

Output:

```
1 2 3
```

Program 16

Program for implementing FCFS scheduling algorithm.

16.1

Problem Definition

Program for implementing FCFS scheduling algorithm.

Problem Description

The First Come First Serve (FCFS) CPU Scheduling is the simplest algorithm. As the name indicates, the process that requests the CPU first, is allocated the CPU. The implementation of the FCFS policy is easily managed by a FIFO queue. When a process enters the queue (ready queue), its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The average waiting time under the FCFS policy, however, is often quite long. The FCFS Scheduling Algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

Algorithm

Step 1 : Initially, we enter the number of processes and the process time.

Step 2 : We then, initialize starting weight time and total weighting time to zero. Initial turnaround time is calculated by adding pt (process/burst time) and wt (waiting time).

Step 3 : The consequent waiting time (and total wt) and turnaround time (and total ta) are calculated using a conditional for loop.

Wt(pi) = wt(pi-1) + tat(pi-1) (i.e wt of current process = wt of previous process + tat of previous process)

tat(pi) = wt(pi) + bt(pi) (i.e tat of current process = wt of current process + bt of current process)

Step 4 : calculate the total and average waiting time and turnaround time

Step 5 : Various print statements are given to display the above calculated times and then, program is terminated.

Problem Validation

Input:

Enter number of processes and their process times.

```
If n = 4
     Burst time for process P1 (in ms) : 10
     Burst time for process P2 (in ms) : 4
     Burst time for process P3 (in ms) : 11
     Burst time for process P4 (in ms) : 6

Pt1=10 Pt2= 4, and pt3=5
```

Output:

The waiting time & turnaround time for processes based on FCFS scheduling was computed and the average waiting time was determined.

```
Process     B-Time     T-Time       W-Time
 P1           10         10            0
 P2            4         14           10
 P3           11         25           14
 P4            6         31           25

Average waiting time:         12.25ms

Average turnaround time:   20.00ms
```

Program for implementing SJF scheduling algorithm.

16.2

Problem Definition

Program for implementing SJF scheduling algorithm.

Problem Description

CPU scheduler will decide which process should be given the CPU for its execution. For this it use different algorithm to choose among the processes. One among those algorithms is SJF algorithm. In this algorithm the CPU will be allocated to the process which has less burst time and after finishing its request only it will allow CPU to execute next other process.

Algorithm

Step 1 : Start the process

Step 2 : Get the number of processes to be inserted

Step 3 : Sort the processes according to the burst time and allocate the one with shortest burst to execute first

Step 4 : If two processes have same burst length then FCFS algorithm is used

Step 5 : Calculate the total and average waiting time and turnaround time

Step 6 : Display the values

Step 7 : Stop the process

Problem Validation

Input:

```
Enter no. of process : 5
Burst time for process P1 (in ms) : 10
Burst time for process P2 (in ms) :  6
Burst time for process P3 (in ms) :  5
Burst time for process P4 (in ms) :  6
Burst time for process P5 (in ms) :  9
```

Output:

The waiting time & turnaround time for processes based on FCFS scheduling was computed and the average waiting time was determined.

```
Process      B-Time     T-Time       W-Time
 P3            5          5            0
 P2            6         11            5
 P4            6         17           11
 P5            9         26           17
 P1           10         36           26

Average waiting time:         11.80ms

Average turn around time:  19.00ms
```

Program for implementing priority scheduling algorithm.

16.3

Problem Definition

Program for implementing priority scheduling algorithm.

Problem Description

The Priority scheduling algorithm is one of the CPU scheduling algorithms. The program executes the processes based on their priorities. CPU scheduler will decide which process should be given the CPU for its execution depending on the priority of the process.

Algorithm

Step 1 : Start the process

Step 2 : Get the number of processes to be inserted

Step 3 : Get the corresponding priority of processes

Step 4 : Sort the processes according to the priority and allocate the one with highest priority to execute first

Step 5 : If two processes have same priority then FCFS scheduling algorithm is used

Step 6 : Calculate the total and average waiting time and turnaround time

Step 7 : Display the values

Step 8 : Stop the process

Problem Validation

Input:

```
Enter no. of process: 3

Process    B-time    Priority
  P1         15         2
  P2          5         1
  P3         10         3
```

Output:

```
PNO    BTIME    PRIORITY  WTIME  TTIME
 2       5        1        0       5
 1      15        2        5      20
 3      10        3       20      30

The average waiting time is: 8.333333

The average turnaround time is: 18.333334
```

Program for implementing Round Robin scheduling algorithm.

16.4

Problem Definition

Program for implementing Round Robin scheduling algorithm.

Problem Description

CPU scheduler will decide which process should be given the CPU for its execution .For this it use different algorithm to choose among the process .one among those algorithms is Round robin algorithm. In this algorithm we are assigning some time slice .The process is allocated according to the time slice ,if the process service time is less than the time slice then process itself will release the CPU voluntarily .The scheduler will then proceed to the next process in the ready queue .If the CPU burst of the currently running process is longer than time quantum ,the timer will go off and will cause an interrupt to the operating system .A context switch will be executed and the process will be put at the tail of the ready queue.

Algorithm

Step 1 : Start the process

Step 2 : Get the number of elements to be inserted

Step 3 : Get the value for burst time for individual processes

Step 4 : Get the value for time quantum

Step 5 : Make the CPU scheduler go around the ready queue allocating CPU to each process for the time interval specified

Step 6 : Make the CPU scheduler pick the first process and set time to interrupt after quantum. And after it's expiry dispatch the process

Step 7 : If the process has burst time less than the time quantum then the process is released by the CPU Step 8 : If the process has burst time greater than time quantum

then it is interrupted by the OS and the process is put to the tail of ready queue and the schedule selects next process from head of the queue

Step 9 : Calculate the total and average waiting time and turnaround time

Step 10 : Display the results

Step 11 : Stop the process

Problem Validation

Input:

```
Enter no. of process: 5
Burst time for process P1: 10
Burst time for process P2: 29
Burst time for process P3: 3
Burst time for process P4: 7
Burst time for process P5: 12
Enter the time slice (in ms): 10
```

Output:

```
Gantt  chart
--------------------------------------------------------------- --
P1   | P2   | P3   |    P4    |  P5   |  P2   |    P5   |     P2    |
--------------------------------------------------------------- --
0    10     20     23        30       40        50       52          61

   PNO      BTIME       WTIME         TTIME
    P1        10          0             10
    P2        29         32             61
    P3         3         20             23
    P4         7         23             30
    P5        12         40             52

The average waiting time is: 23.00 ms

The average turnaround time is: 35.20 ms
```

Program 17

To implement page replacement algorithms using FIFO

17.1

Problem Definition

To implement page replacement algorithms using FIFO

Problem Description

To execute a program all the pages of the program should be loaded in to the main memory. By using virtual memory concept only few pages of a program are loaded and if the required page is not available in the main memory (when it is in the execution state) then page fault occur. We have to load the required page into the main memory , if no space in the main memory then replace the frame which is loaded first into memory with required the page is the FIFO page replacement algorithm. This program calculates the number of page faults by using FIFO concept.

Algorithm

```
Step 1 : Get length of the reference string, say l
Step 2 : Get reference string and store it in an array, say rs.
Step 3 : Get number of frames, say nf.
Step 4 : Initialize frame array upto length nf to -1
Step 5 : Initialize position of the oldest page, say j to 0.
Step 6 : Initialize no. of page faults, say count to 0.
Step 7 : For each page in reference string in the given order, examine:
    1. Check whether page exist in the frame array
    2. If it does not exist then
        a. Replace page in position j.
        b. Compute page replacement position as (j+1) modulus nf.
        c. Increment count by 1.
        d. Display pages in frame array.
Step 8 : Print count
Step 9 : Stop
```

Problem Validation

Input:

```
Enter length of ref. string: 20

Enter reference string: 1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6

Enter number of frames: 5

\vspace{0.1in}\noindent {\bf Output:}

Ref. str Page frames
1          1  -1 -1  -1  -1
2          1   2 - 1 -1  -1
3          1   2   3  -1  -1
4          1   2   3   4  -1
2
1
5          1   2   3   4   5
6          6   2   3   4   5
2
1          6   1   3   4   5
2          6   1   2   4   5
3          6   1   2   3   5
7          6   1   2   3   7
6
3
2
1
2
3
6

Total no. of page faults: 10
```

17.2

Problem Definition

To implement page replacement algorithms using LRU

Problem Description

To execute a program all the pages of the program should be loaded in to the main memory. By using virtual memory concept only few pages of a program are loaded and if the required page is not available in the main memory (when it is in the execution state) then page fault occur. We have to load the required page into the main memory , if no space in the main memory then replace the frame which is least recently used - LRU page replacement algorithm. This program calculates the number of page faults by using FIFO concept.

Algorithm

Step 1 : Get length of the reference string, say len.

Step 2 : Get reference string and store it in an array, say rs.

Step 3 : Get number of frames, say nf.

Step 4 : Create access array to store counter that indicates a measure of recent usage.

Step 5 : Create a function arrmin that returns position of minimum of the given array.

Step 6 : Initialize frame array up to length nf to -1

Step 7 : Initialize position of the page replacement, say j to 0.

Step 8 : Initialize freq to 0 to track page frequency

Step 9 : Initialize no. of page faults, say count to 0.

Step 10 : For each page in reference string in the given order, examine:

1. Check whether page exist in the frame array.
2. If page exist in memory then i. Store incremented freq for that page position in access array.
3. If page does not exist in memory then
   a. Check for any empty frames.
   b. If there is an empty frame, Assign that frame to the page Store incremented freq for that page position in access array. Increment count.
   c. If there is no free frame then determine page to be replaced using arrmin function. Store incremented freq for that page position in access array. Increment count.

d. Display pages in frame array.

Step 8 : Print count

Step 9 : Stop

Problem Validation

Input:

```
Enter length of ref. string: 20

Enter reference string: 1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6

Enter number of frames: 5
```

Output:

```
Ref. str     Page frames
1 1  -1  -1  -1  -1
2 1   2  -1  -1  -1
3 1   2   3  -1  -1
4 1   2   3   4  -1

2
1
5 1   2   3    4    5
6 1   2   6    4    5
2
1
2
3         1   2   6   3    5
7 1   2   6   3    7
6
3
2
1
2
3

6 Total no. of page faults: 8
```

Program 18

Producer-Consumer problem using shared memory

Problem Definition

Producer-Consumer problem using shared memory

Problem Description

The general statement is this: there are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time

The problem is to make sure that the producer won't try to add data in to the buffer if it's full and that the consumer won't try to remove data from an empty buffer

Algorithm

Step 1 : Create a shared memory segment BUFSIZE of size 1 and attach it.

Step 2 : Obtain semaphore id for variables empty, mutex and full using semget function.

Step 3 : Create semaphore for empty, mutex and full as follows: Declare semun, a union of specific commands. The initial values are: 1 for mutex, N for empty and 0 for full Use semctl function with SETVAL command

Step 4 : Create a child process using fork system call. Make the parent process to be the producer Make the child process to the consumer

Step 5 : The producer produces 5 items as follows: Call wait operation on semaphores empty and mutex using semop() Gain access to buffer and produce data for consumption Call signal operation on semaphores mutex and full using semop().

Step 6 : The consumer consumes 5 items as follows: Call wait operation on semaphores full and mutex using semop function. Gain access to buffer and consume the available data. Call signal operation on semaphores mutex and empty using semop function.

Step 7 : Remove shared memory from the system using shmctl with IPC_RMID argument

Step 8 : Stop

Problem Validation

Input:

Execute the steps of the algorithm in the given order.

Output:

```
Enter data for producer to produce: 5
Enter data for producer to produce: 8
Consumer consumes data 5
Enter data for producer to produce: 4
Consumer consumes data 8
Enter data for producer to produce: 2
Consumer consumes data 4
Enter data for producer to produce: 9
Consumer consumes data 2
Consumer consumes data 9
```

Program 19

To implement Readers-Writers problem using message passing

Problem Definition

To implement Readers-Writers problem using message passing

Problem Description

This problem is a generalization of the mutual exclusion problem. There are two types of processes: The readers that only read the information and thus can simultaneously access the critical section; The writers that modify (write) the data and thus must have access in strict mutual exclusion. Thus, may simultaneous have access to the critical section either

- A single writer and no readers, or
- Several readers and no writers.

```
The structure of a writer process:
wait(wrt); //waiting is performed
signal(wrt);
 The structure of a reader process:
  wait(mutex);
readcount++;
if(readcount == 1)
   wait(wrt);
signal(mutex); // reading is performed
wait(mutex);
read count - -;
if(readcount == 0)
  signal(wrt);
signal(mutex);
```

Algorithm

Step 1 : Reader will run after Writer because of read semaphore.

Step 2 : Writer will stop writing when the write semaphore has reached 0.

Step 3 : Reader will stop reading when the read semaphore has reached 0.

Problem Validation

Input:

Execute the steps of the algorithm in the given order.

Output:

Writer 0 is here

Writer 1 is here

Program 20

To implement Dining Philosopher's Problem using semaphore

Problem Definition

To implement Dining Philosopher's Problem using semaphore

Problem Description

There are 5 philosophers who spend their lives thinking and eating. The philosophers share common circular table surrounded by 5 chairs, each for one philosopher, and the table is laid only with 5 chopsticks. The dining philosopher's problem is considered as a class synchronisation problem because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock and starvation free-manner. One simple solution is to represent each chopstick by a semaphore. A philosopher tries to grab the chopstick by executing a wait operation on that semaphore, and releases her chopsticks by executing the signal operation on the appropriate semaphores.

Algorithm

Step 1 : The function is compared with process id and then, the philosophers 0, 1,2,3,4 are displayed accordingly.

Step 2 : Each philosopher is then compared using a wait and signal operations.

Step 3 : When operation is 1, then it goes under signal state and increments the value and displays the output.

Step 4 : When operation is -1, then it goes under wait state and decrements the value and displays the output.

Step 5 : All the values are returned and each philosopher is displayed accordingly. The program terminates.

Problem Validation

Input:
Execute the steps of the algorithm in the given order.

Output:

```
Philosopher 0 is hungry
Philosopher 0 is eating
Philosopher 1 is hungry
Philosopher 1 is eating
Philosopher 2 is hungry
Philosopher 2 is eating
Philosopher 3 is hungry
Philosopher 3 is eating
Philosopher 4 is hungry
Philosopher 4 is eating
```

Program 21

To implement deadlock avoidance and prevention.

Problem Definition

To implement deadlock avoidance and prevention.

Problem Description

To implement deadlock avoidance & Prevention by using Banker's Algorithm. Banker's Algorithm: When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Data structures

- n-Number of process, m-number of resource types.
- Available: Available[j]=k, k – instance of resource type Rj is available.
- Max: If max[i, j]=k, Pi may request at most k instances resource Rj.
- Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj
- Need: If Need[I, j]=k, Pi may need k more instances of resource type Rj,
- Need[I, j]=Max[I, j]-Allocation[I, j];

Safety Algorithm

1. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] = False.
2. Find an i such that both Finish[i] =False and Need•<=Work If no such I exists go to step 4.
3. work=work+Allocation, Finish[i] =True;
4. if Finish[1]=True for all I, then the system is in safe state. Resource request algorithm: Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.

1. If Request<=Need I go to step 2. Otherwise raise an error condition.
2. if Request<=Available go to step 3. Otherwise Pi must since the resources are available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows;

   Available = Available-Request I;
   Allocation I = Allocation+Request I;
   Need i = Need i-Request I;

   If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state is restored.

Algorithm

Step 1 : Start the program.

Step 2 : Get the values of resources and processes.

Step 3 : Get the avail value.

Step 4 : After allocation find the need value.

Step 5 : Check whether it's possible to allocate.

Step 6 : If it is possible then the system is in safe state.

Step 7 : Else system is not in safety state.

Step 8 : If the new request comes then check that the system is in safety.

Step 9 : or not if we allow the request.

Step 10 : stop the program.

Problem Validation

Input:

Execute the steps of the algorithm in the given order.

Output:

```
Enter the no of processes:
2
Enter the no of resources:
3
Enter the claim for each process:
For process I : 2 4 5
For process II: 2 5 3
Enter the total no of each resource : 5 5 2
Available resource is :
Claim matrix: allocation matrix:
245 123
253 234
The system is in an unsafe state!!
```

Program 22

Factorial of a given number using Shell Program

22.1

Problem Definition

To find the factorial of a given number using Shell Program

Problem Description

The real power of computer is its ability to do the work for us. To get the computer to do that, we use the power of shell. To automate things, we write scripts. Scripts are collections of commands that are stored in a file. The shell can read this file and act on commands as if they were typed at the keyboard. The shell also provides a variety of useful programming features to make our scripts truly powerful. The program finds the factorial of a given number. Ex : factorial of 5 = 5*4*3*2

Syntax of while loop:

```
while [condition]
do
____________
____________-
____________-
done
```

Algorithm

Step 1 : Read n and make i=1, f=1.

Step 2 : Compare ‘i’ with ‘n’ and then, multiply ‘f’ with ‘i’

Step 3 : Later, ‘i’ value is incremented by 1. The loop continues until i=n and final output of factorial is displayed.

Step 4 : The program terminates.

Problem Validation

Input:

Enter n = 5

Output:

The factorial of 5 is 120.

The given Number is Palindrome or not.

22.2

Problem Definition

To find whether the given number is palindrome or not.

Problem Description

Palindrome means if a number is reversed then we will get the original number only. Ex: 121. The reverse of 121 is 121 only then the number is palindrome.

Algorithm

Step 1 : Read n

Step 2 : n = num, rev = 0

Step 3 :

```
while (num > 0)
{
      dig = num % 10;
      rev = rev * 10 + dig;
      num = num / 10;
}
```

Step 4 : if n=num diplay given number is palindrome else not palindrome.

Step 5 : Stop.

Problem Validation

Input:

Enter n = 121

Output:

The given number is palindrome.

To find the Net Salary of an Employee.

22.3

Problem Definition

To find the net salary of an employee.

Problem Description

The shell provides a variety of useful programming features to make our scripts truly powerful. This is a program in which the net salary of an employee is calculated. It uses simple arithmetic, logical and relational operations. We try to find da, hra, gross, pf, ded, net by using basic and it data. It helps us in solving large number of records and storing heavy data.

Algorithm

```
Step 1 : Read the 'basic' and 'it' value.
Step 2 : Find the DA, HRA and PF based on BASIC
Step 3 : GROSS = DA + HRA + BASIC
Step 4 : DED= PF + IT + PT.
Step 5 : Find NET = GROSS – DED and display the NET output.
Step 5 : The program terminates.
```

Problem Validation

Input:

Enter basic = 30000

Output:

NET = 56200

Annexure – I

List of programs according to O.U. curriculum

Code: CS232	**OS LAB**	
Instruction	**3**	**Periods per week**
Duration of University Examination	**3**	**Hours**
University Examination	**50**	**Marks**
Sessional	**25**	**Marks**

1. Printing the file flags for specified descriptor.
2. Print type of file for each command line arguments
3. Recursively descend a directory hierarchy counting file types
4. Program using process related system calls
5. Implement CPU scheduling algorithms (a) Round Robin (b) SJF (c) FCFS
6. Implement page replacement algorithms (a) FIFO (b) LRU
7. Echo server using pipes
8. Echo server using messages
9. Producer-Consumer problem using shared memory
10. Readers – Writers problem using message passing
11. Dinning philosophers problem using semaphores
12. Bankers algorithm for Deadlock detection and avoidance
13. Program using file locking
14. Programs using LINUX shell scripts

Prescribed Textbooks :

1. Operating Systems Concepts, 8th edition, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne; Wiley, ISBN 0-470- 12872-0, 2010.

Reference Textbooks :

1. Operating Systems: Internals and Design Principles, 6th edition, William Stallings; Prentice Hall, ISBN-10: 0136006329, 2009.
2. Operating Systems, 3rd edition, Gary Nutt; Pearson/Addison Wesley, ISBN 0-201-77344-9, 2004.
3. Modern Operating Systems, 3rd edition, Andrew S. Tanenbaum; Prentice Hall, ISBN-10: 0-13-600663-9, 2008.