As Per New VTU Syllabus w.e.f 2018-19
Choice Based Credit System(CBCS)

SUNSTAR

SUNSTAR EXAM SCANNER

UNIX PROGRAMMING

(V SEM.B.E. CSE/ISE)

SYLLABUS

UNIX PROGRAMMING

[AS PER CHOICE BASED CREDIT SYSTEM (CBCS) SCHEME]
(EFFECTIVE FROM THE ACADEMIC YEAR 2018 - 2019)

Subject Code	18CS56	CIE Marks	40
Number of Contact Hours/Week	3:0:0	SEE Marks	60
Total Number of Contact Hours	40	Exam Hours	03

MODULE 1

Introduction: Unix Components/Architecture. Features of Unix. The UNIX Environment and UNIX Structure, Posix and Single Unix specification. General features of Unix commands/ command structure. Command arguments and options. Basic Unix commands such as echo, printf, ls, who, date,passwd, cal, Combining commands. Meaning of Internal and external commands. The type command: knowing the type of a command and locating it. The root login. Becoming the super user: su command.

Unix files: Naming files. Basic file types/categories. Organization of files. Hidden files. Standard directories. Parent child relationship. The home directory and the HOME variable. Reaching required files- the PATH variable, manipulating the PATH, Relative and absolute pathnames. Directory commands – pwd, cd, mkdir, rmdir commands. The dot (.) and double dots (..) notations to represent present and parent directories and their usage in relative path names. File related commands – cat, mv, rm, cp, wc and od commands.

MODULE 2

File attributes and permissions: The ls command with options. Changing file permissions: the relative and absolute permissions changing methods. Recursively changing file permissions. Directory permissions.

The shells interpretive cycle: Wild cards. Removing the special meanings of wild cards. Three standard files and redirection. Connecting commands: Pipe. Basic and Extended regular expressions. The grep, egrep. Typical examples involving different regular expressions.

Shell programming: Ordinary and environment variables. The .profile. Read and readonly commands. Command line arguments. exit and exit status of a command. Logical operators for conditional execution. The test command and its shortcut. The if, while, for and case control statements. The set and shift commands and handling positional parameters. The here (<<) document and trap command. Simple shell program examples.

MODULE 3

UNIX File APIs: General File APIs, File and Record Locking, Directory File APIs, Device File APIs, FIFO File APIs, Symbolic Link File APIs.

UNIX Processes and Process Control:

The Environment of a UNIX Process: Introduction, main function, Process Termination, Command-Line Arguments, Environment List, Memory Layout of a C Program, Shared Libraries, Memory Allocation, Environment Variables, setjmp and longjmp Functions, getrlimit, setrlimit Functions, UNIX Kernel Support for Processes.

Process Control: Introduction, Process Identifiers, fork, vfork, exit, wait, waitpid, wait3, wait4 Functions, Race Conditions, exec Functions

MODULE 4

Changing User IDs and Group IDs, Interpreter Files, system Function, Process Accounting, User Identification, Process Times, I/O Redirection.

Overview of IPC Methods, Pipes, popen, pclose Functions, Coprocesses, FIFOs, System V IPC, Message Queues, Semaphores.

Shared Memory, Client-Server Properties, Stream Pipes, Passing File Descriptors, An Open Server-Version 1, Client-Server Connection Functions.

MODULE 5

Signals and Daemon Processes: Signals: The UNIX Kernel Support for Signals, signal, Signal Mask, sigaction, The SIGCHLD Signal and the waitpid Function, The sigsetjmp and siglongjmp Functions, Kill, Alarm, Interval Timers, POSIX.lb Timers. Daemon Processes: Introduction, Daemon Characteristics, Coding Rules, Error Logging, Client-Server Model.

Fifth Semester B.E. Degree Examination
CBCS - Model Question Paper - 1
UNIX PROGRAMMING

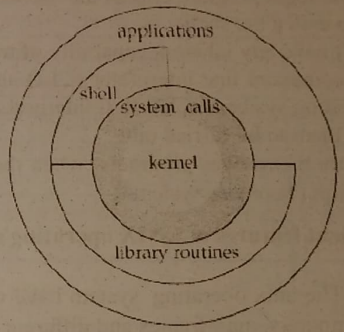
Time: 3 hrs. Max. Marks: 100

Note : *Answer any FIVE full questions, selecting ONE full question from each module.*

Module - 1

1. a. With a neat diagram, explain the architecture of UNIX operating system. **(08 Marks)**

Ans.



There are two important divisions in UNIX operating system architecture.

1. Kernel
2. Shell

In simple words you can say –

- Kernal – interacts with the machine's hardware
- Shell – interacts with the user

THE KERNEL: The kernel of UNIX is the hub (or core) of the UNIX operating system. Kernel is a set of routines mostly written in C language.

User programs that need to access the hardware (like hard disk or terminal) use the services of the Kernel, which performs the job on the user's behalf.

User interacts with the Kernal by using System calls. Kernel allocates memory and time to programs and handles the file store and communications in response to system calls.

As an illustration of the way that the unix shell and the kernel work together, suppose a user types mv myfile myfile1 (which has the effect of renaming the file myfile). The unix shell searches the file store for the file containing the program mv, and then requests the kernel, through system calls, to execute the program mv on myfile. When the process mv myfile has finished running, the unix shell then returns the UNIX prompt to the user, indicating that it is waiting for further commands.

Sunstar Exam Scanner

SOME OTHER FUNCTIONS PERFORMED BY THE KERNEL IN UNIX SYSTEM ARE:

1. Managing the machine's memory and allocating it to each process and decides their priorities.
2. Scheduling the work done by the CPU so that the work of each user is carried out as efficiently as is possible.
3. Organizing the transfer of data from one part of the machine to another.
4. Accepting instructions from the unix shell and carrying them out.
5. Enforcing the access permissions that are in force on the file system

THE SHELL:

UNIX Shell acts as a medium between the user and the kernel in unix system. When a user logs in, the login program checks the username and password and then starts another program called the shell.

Computers don't have any inherent capability of translating commands into action. This requires a command line interpreter (CLI) and this is handled by the "Outer Part" of the operating system i.e. Shell. It interprets the commands the user types in and arranges for them to be carried out.

The commands are themselves programs: when they terminate, the shell gives the user another prompt (% on our systems).

b. Describe the salient features of UNIX operating system. (08 Marks)

Ans.

1. **Popularity :** The unix operating system have wide range of computing power from microcomputers, mainframes and different manufacture's machine.
2. **Portability :** It is easier to read, understand, change and move to other machines because it is written in high-level language. The code can be changed and complied on a new machine. Users can then choose from a wide variety of hardware vendors without being locked in with a particular vendor.
3. **Machine-independence :** The system is machine-independent. So , it is easier to write applications that can run on micros, mins and mainframes because the system hides the machine architecture from the user.
4. **Multi-User Operations :** UNIX is a multi-user system designed to support a group of users simultaneously. The system allows for the sharing of processing power and peripheral resources, white at the same time providing excellent security features.
5. **Hierarchical File System :** UNIX uses a hierarchal file structure to store information. This structure has the maximum flexibility in grouping information in a way that reflects its natural state. It allows for easy maintenance and efficient implementation.
6. **UNIX shell :** UNIX has a simple user interface called the shell that has the power to provide the services that the user wants. It protects the user from having to know the intricate hardware details.
7. **Pipes and Filters :** UNIX has facilities called Pipes and Filters which permit the user to create complex programs from simple programs.

Sunstar Exam Scanner

8. **Utilities :** UNIX has over 200 utility programs for various functions. New utilities can be built effortlessly by combining existing utilities.
9. **Software Development Tools :** UNIX offers an excellent variety of tools for software development for all phases, from program editing to maintenance of software.

c. Differentiate between internal and external commands in UNIX with suitable examples. (04 Marks)

Ans.

Base	Internal Command	External Command
Installation	Internal commands are built in Command that pre-contained to the file "Command.com"	External Commands are not contained to the file "Command.com"
Independence	Internal Commands are Independent.	External Command are Independent.
Eexecution requirement	Internal commands don't require separate process.	External commands require separate process.
Executed by	Directly by shell	By kernal
Path requirement	Internal Command don't require Path.	External Command require Path.
Execution Speed	Very high	Slower than Internal Command.
Shell	Internal Commands are part of shell.	External Commands are not part of Shell.
Load	Internal Command always load on primary memory	External Command only loaded by user request for execution.
Example	DIR, VLO, DEL, MD, CD, REN, DATE, TIME, PATH etc.	EDIT, BACKUP, MORE, XCOPY.FC, DISKCOPY, MODE etc.

OR

2. a. Write a notes on: i) POSIX.1 FIPS standard ii) X/open standard (10 Marks)

Ans. i) POSIX.1 FIPS standard:

FIPS stands for Federal Information Processing Standard. This standard was deveioped by National Institute of Standards and Technology. The latest version of this standard, FIPS 151-1, is based on the POSIX.1-1998 standard. The FIPS standard is a restriction of the POSIX.1-1998 standard, Thus a FIPS 151-1 conforming system is also POSIX.1-1998 conforming, but not vice versa.

FIPS 151-1 conforming system requires following features to be implemented in all FIPS conforming systems.

Sunstar Exam Scanner

_POSIX_JOB_CONTROL	_POSIX_JOB_CONTROL must be defined.
_POSIX_SAVED_IDS	_POSIX_SAVED_IDS must be defined.
_POSIX_CHOWN_RESTRICTED	_POSIX_CHOWN_RESTRICTED must be defined and its value is not -1, it means users with special privilege may change ownership of any files on a system.
_POSIX_NO_TRUNC	If the defined value is -1, any long path name passed to an API is silently truncated to NAME_MAX bytes, otherwise error is generated.
_POSIX_VDISABLE	POSIX_VDISABLE must be defined and its value is not -1.
_POSIX_NO_TRUNC	Must be defined and its value is not -1, Long path name is not support.
NGROUP_MAX	Symbol's value must be at least 8.
The read and write API should return the number of bytes that have been transferred after the APIs have been	
The group ID of a newly created file must inherit the group ID of its containing directory.	

ii) X/open Standard

The X/Open DTP standard defines how transaction processing is performed in a distributed, open environment. In this environment, three logical components interact to execute global transactions (logical transactions that may span multiple hardware and software platforms):

- **Resource manager**

The resource manager (RM) manages access to data (and possibly other shared resources). In an Ingres DTP installation, the resource manager is an Ingres database server acting in combination with Ingres DTP library routines linked into the application.

- **Transaction manager**

The transaction manager (TM) oversees the execution of global transactions. The transaction manager performs the following functions:

a) It accepts global transaction start, commit, and rollback calls from the application program. (Transaction rollback can also be initiated by the transaction manager itself or by the resource manager.)

b) It directs resource managers to start, end, prepare, commit, and rollback global transactions. To communicate with resource managers, the transaction manager calls XA routines provided by the resource managers.

- **Application program**

The application program performs the following functions:

a) It interacts with the end-user.

b) It notifies the transaction manager when it wants to begin, commit, or abort a transaction. To communicate with the TM, the application program calls routines supplied by the TP vendor.

c) It performs database access. To interact with an Ingres database server, the application program uses Ingres embedded SQL.

Sunstar Exam Scanner

b. Explain the common characteristics of API and describe the error status code and their meaning. **(10 Marks)**

Ans. When the APIs execution completes, the user process is switched back to the user mode. This context switching for each API call ensures that process access kernels data in a controlled manner and minimizes any chance of a runway user application may damage an entire system. So in general calling an APIs is more time consuming than calling a user function due to the context switching. Thus for those time critical applications, user should call their system APIs only if it is necessary.

An APIs common Characteristics

Most system calls return a special value to indicate that they have failed. The special value is typically -1, a null pointer, or a constant such as EOF that is defined for that purpose.

To find out what kind of error it was, you need to look at the error code stored in the variable errno. This variable is declared in the header file errno.h as shown below.

volatile int errno

- The variable errno contains the system error number.

void perror (const char *message)

- The function perror is declared in stdio.h.

Following table shows Some Error Codes and their meaning:

Errors	Meaning
EPERM	API was aborted because the calling process does not have the super user privilege.
EINTR	An APIs execution was aborted due to signal interruption.
EIO	An Input/Output error occurred in an APIs execution.
ENOEXEC	A process could not execute program via one of the Exec API.
EBADF	An API was called with an invalid file descriptor.
ECHILD	A process does not have any child process which it can wait on.
EAGAIN	An API was aborted because some system resource it is requested was temporarily unavailable. The API should call again later.
ENOMEM	An API was aborted because it could not allocate dynamic memory.
EACCESS	The process does not have enough privilege to perform the operation.
EFAULT	A pointer points to an invalid address. .
EPIPE	An API attempted to write data to a pipe which has no reader.
ENOENT	An invalid file name was specified to an API.

Module - 2

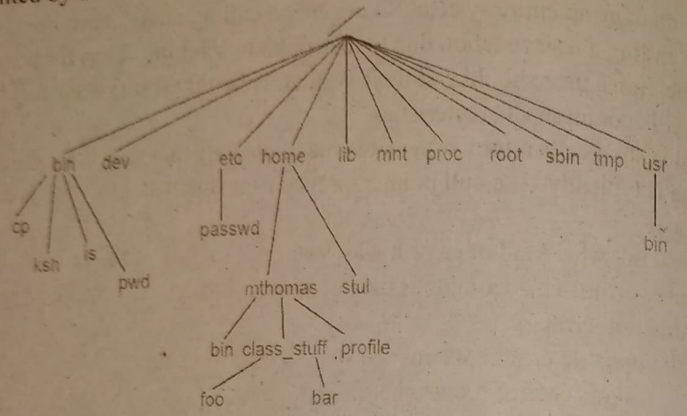
3. a. What is a file? Explain different file types available in UNIX and POSIX system. Also write the commands to create all the files. **(10 Marks)**

Ans. Unix file system is a logical method of organizing and storing large amounts of information in a way that makes it easy to manage. A file is a smallest unit in which the information is stored. Unix file system has several important features. All data in

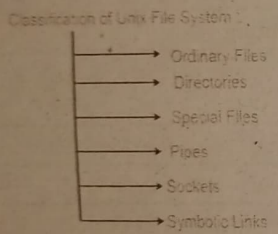
Sunstar Exam Scanner

Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the file system.

Files in Unix System are organized into multi-level hierarchy structure known as a directory tree. At the very top of the file system is a directory called "root" which is represented by a "/". All other files are "descendants" of root.



Types of Unix files – The UNIX files system contains several different types of files :



1. Ordinary files – An ordinary file is a file on the system that contains data, text, or program instructions.

- Used to store your information, such as some text you have written or an image you have drawn. This is the type of file that you usually work with.
- Always located within/under a directory file.
- Do not contain other files.
- In long-format output of ls -l, this type of file is specified by the "-" symbol.

2. Directories – Directories store both special and ordinary files. For users familiar with Windows or Mac OS, UNIX directories are equivalent to folders. A directory file contains an entry for every file and subdirectory that it houses. If you have 10 files in a directory, there will be 10 entries in the directory. Each entry has two components.

1) The Filename
2) A unique identification number for the file or directory (called the inode number)

- Branching points in the hierarchical tree.
- Used to organize groups of files.
- May contain ordinary files, special files or other directories.

Sunstar Exam Scanner

- Never contain "real" information which you would work with (such as text). Basically, just used for organizing files.
- All files are descendants of the root directory, (named /) located at the top of the tree.

In long-format output of ls –l , this type of file is specified by the "d" symbol.

3. Special Files – Used to represent a real physical device such as a printer, tape drive or terminal, used for Input/Output (I/O) operations. Device or special files are used for device Input/Output(I/O) on UNIX and Linux systems. They appear in a file system just like an ordinary file or a directory.

On UNIX systems there are two flavors of special files for each device, character special files and block special files :

- When a character special file is used for device Input/Output(I/O), data is transferred one character at a time. This type of access is called raw device access.
- When a block special file is used for device Input/Output(I/O), data is transferred in large fixed-size blocks. This type of access is called block device access.

4. Pipes – UNIX allows you to link commands together using a pipe. The pipe acts a temporary file which only exists to hold data from one command until it is read by another. A Unix pipe provides a one-way flow of data. The output or result of the first command sequence is used as the input to the second command sequence. To make a pipe, put a vertical bar (|) on the command line between two commands. For example: who | wc -l

In long-format output of ls –l , named pipes are marked by the "p" symbol.

5. Sockets – A Unix socket (or Inter-process communication socket) is a special file which allows for advanced inter-process communication. A Unix Socket is used in a client-server application framework. In essence, it is a stream of data, very similar to network stream (and network sockets), but all the transactions are local to the filesystem.

In long-format output of ls -l, Unix sockets are marked by "s" symbol.

6. Symbolic Link – Symbolic link is used for referencing some other file of the file system. Symbolic link is also known as Soft link. It contains a text form of the path to the file it references. To an end user, symbolic link will appear to have its own name, but when you try reading or writing data to this file, it will instead reference these operations to the file it points to. If we delete the soft link itself , the data file would still be there. If we delete the source file or move it to a different location, symbolic file will not function properly.

In long-format output of ls –l , Symbolic link are marked by the "l" symbol (that's a lower case L).

b. Assume that you are in /home/Kumar, which of these commands will work when executed in sequence? Explain the proper reasons.
mkdir a/b/c→mkdir a a/b
mkdir a a/b a/b/c→rmdir a/b/c→rmdir a a/b→ mkdir a/p a/q a/r
Draw the final tree structure for directory 'a' **(04 Marks)**

Sunstar Exam Scanner

Ans. i) mkdir a/b/c doesn't execute, because the mkdir is trying to create subdirectories (b and c) inside 'a' which is not available
ii) mkdir a a/b successfully command will be executed, because first directory 'a' will be created and inside 'a' subdirectory 'b' will be created.
iii) mkdir a a/b a/b/c successfully command will be executed, because first directory 'a' will be created and inside subdirectory 'b' will be created. Once subdirectory 'b' created inside 'a', 'c' will be created Inside 'b'
iv) rmdir a/b/c will remove subdirectories 'b' and 'c' inside 'a'
v) rmdir a a/b this command tries to remove 'b' which is not present inside 'a' so unsuccessful execution of command
vi) mkdir a/p a/q a/r creates subdirectories 'p', 'q', 'r' inside directory 'a'

```
|-home
|--Kumar
|---a
|----p
|----q
|----r
```

c. Explain the following commands with an example i) cd ii) pwd iii)rmdir iv) wc (06 Marks)

Ans. **i) cd:** cd command in linux known as change directory command. It is used to change current working directory.
$ cd [directory]
To move inside a subdirectory : to move inside a subdirectory in linux we use
$ cd [directory_name]
ii) pwd: pwd stands for Print Working Directory. It prints the path of the working directory, starting from the root.
pwd is shell built-in command(pwd) or an actual binary(/bin/pwd).
$PWD is an environment variable which stores the path of the current directory.
This command has two flags.
pwd -L: Prints the symbolic path.
pwd -P: Prints the actual path.
A) Built-in pwd (pwd):

```
File Edit View Terminal Tabs Help
shital@debian:~/logs$ pwd
/home/shital/logs
shital@debian:~/logs$ pwd -L
/home/shital/logs
shital@debian:~/logs$ pwd -P
/var/log
shital@debian:~/logs$
```

In the given example the directory /home/shital/logs/ is a symbolic link for a target directory /var/logs/
B) Binary pwd (/bin/pwd):


```
File Edit View Terminal Tabs Help
shital@debian:~/logs$ /bin/pwd
/var/log
shital@debian:~/logs$ /bin/pwd -P
/var/log
shital@debian:~/logs$ /bin/pwd -L
/home/shital/logs
shital@debian:~/logs$
```

The default behavior of Built-in pwd is same as pwd -L.
And the default behavior of /bin/pwd is same as pwd -P.
The $PWD variable value is same as pwd -L.
iii) rmdir
rmdir command is used remove empty directories from the filesystem in Linux. The rmdir command removes each and every directory specified in the command line only if these directories are empty. So if the specified directory has some directories or files in it then this cannot be removed by rmdir command.
Syntax:
rmdir [-p] [-v | –verbose] [–ignore-fail-on-non-empty] directories ...
rmdir -p mydir/mydir1
This will first remove the child directory and then remove the parent directory.
iv) wc:
wc stands for word count. As the name implies, it is mainly used for counting purpose.
• It is used to find out number of lines, word count, byte and characters count in the files specified in the file arguments.
• By default it displays four-columnar output.
• First column shows number of lines present in a file specified, second column shows number of words present in the file, third column shows number of characters present in file and fourth column itself is the file name which are given as argument.
Syntax:
wc [OPTION]... [FILE]...
Let us consider two files having name state.txt and capital.txt containing 5 names of the Indian states and capitals respectively.

```
$ cat state.txt
Andhra Pradesh
Arunachal Pradesh
Assam
Bihar
Chhattisgarh
$ cat capital.txt
Hyderabad
Itanagar
Dispur
Patna
```

Raipur
Passing only one file name in the argument.
$ wc state.txt
5 7 63 state.txt
OR
$ wc capital.txt
5 5 45 capital.txt

OR

4. a. Briefly describe the significance of the seven fields of the 'ls-l' command. (10 Marks)

Ans. $ ls -l
-rw-r----- 1 ramesh team-dev 9275204 Jun 13 15:27 mthesaur.txt.gz

- 1st Character- File TypeFirst character specifies the type of the file. In the example above the hyphen (-) in the 1st character indicates that this is a normal file. Following are the possible file type options in the 1st character of the ls -l output.
  1) Field Explanation
  2) - normal file
  3) d directory
  4) s socket file
  5) l link file
- Field 1 – File Permissions: Next 9 character specifies the files permission. Each 3 characters refers to the read, write, execute permissions for user, group and world In this example, -rw-r—— indicates read-write permission for user, read permission for group, and no permission for others.
- Field 2 – Number of links: Second field specifies the number of links for that file. In this example, 1 indicates only one link to this file.
- Field 3 – Owner: Third field specifies owner of the file. In this example, this file is owned by username 'ramesh'.
- Field 4 – Group: Fourth field specifies the group of the file. In this example, this file belongs to "team-dev' group.
- Field 5 – Size: Fifth field specifies the size of file. In this example, '9275204' indicates the file size.
- Field 6 – Last modified date & time: Sixth field specifies the date and time of the last modification of the file. In this example, 'Jun 13 15:27' specifies the last modification time of the file.
- Field 7 – File name: The last field is the name of the file. In this example, the file name is mthesaur.txt.gz.

ls -l command will display the below result
total 136
drwxr-xr-x 2 root root 4096 Jun 5 15:16 bin
drwxr-xr-x 3 root root 4096 Jul 9 15:25 boot
drwxr-xr-x 9 root root 3240 Jul 17 20:08 dev

Sunstar Exam Scanner

drwxr-xr-x 40 root root 4096 Aug 14 16:51 etc
1st column is --> file type(DIR/Files) and access details for UGO (User Group Others).
2nd column is --> position of the file/dir and Name of the Root (2,9)
3rd Column is --> Name of the Root and file located path (root)
4th Column is --> - do-- (i am not sure)
5th Column is Size of the file and Dir (4096)
6th Column is (Timestamp)Date and time of file / DIR created. (Aug 14 16:51
7th Column is File Directory name (dev,boot)

b. With suitable example, bring out the detailed difference between absolute and relative pathnames. (10 Marks)

Ans. A path is a unique location to a file or a folder in a file system of an OS.A path to a file is a combination of / and alpha-numeric characters.

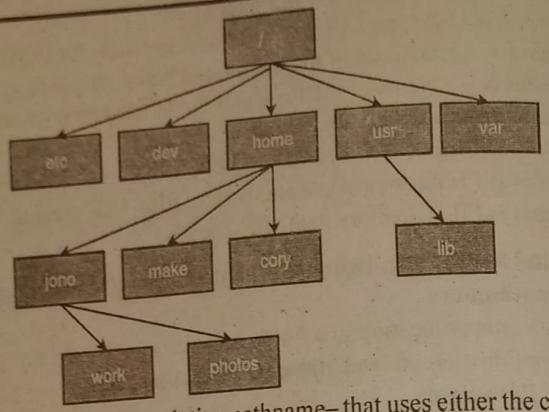
Absolute Path-name

An absolute path is defined as the specifying the location of a file or directory from the root directory(/).
To write an absolute path-name:
Start at the root directory (/) and work down.
Write a slash (/) after every directory name (last one is optional)
For Example : $cat abc.sql
will work only if the fie "abc.sql" exists in your current directory. However, if this file is not present in your working directory and is present somewhere else say in / home/kt , then this command will work only if you will use it like shown below:
cat /home/kt/abc.sql
In the above example, if the first character of a pathname is /, the file's location must be determined with respect to root. When you have more than one / in a pathname, for each such /, you have to descend one level in the file system like in the above kt is one level below home, and thus two levels below root.

Relative path

Relative path is defined as the path related to the present working directly(pwd). It starts at your current directory and never starts with a / .
To be more specific let's take a look on the below figure in which if we are looking for photos then absolute path for it will be provided as /home/jono/photos but assuming that we are already present in jono directory then the relative path for the same can be written as simple photos.

Sunstar Exam Scanner



UNIX offers a shortcut in the relative pathname– that uses either the current or parent directory as reference and specifies the path relative to it. A relative path-name uses one of these cryptic symbols:

.(a single dot) - this represents the current directory.

..(two dots) - this represents the parent directory.

Now, what this actually means is that if we are currently in directory /home/kt/abc and now you can use .. as an argument to cd to move to the parent directory /home/kt as :

```
$pwd
/home/kt/abc
$cd ..          ***moves one level up***
$pwd
/home/kt
```

NOTE:Now / when used with .. has a different meaning ;instead of moving down a level,it moves one level up:

```
$pwd
/home/kt/abc    ***moves two level up***
$cd ../..
$pwd
/home
```

Example of Absolute and Relative Path

Suppose you are currently located in home/kt and you want to change your directory to home/kt/abc. Let's see both the absolute and relative path concepts to do this:

1) Changing directory with relative path concept :

```
$pwd
/home/kt
$cd abc
$pwd
/home/kt/abc
```

2) Changing directory with absolute path concept:

Sunstar Exam Scanner


```
$pwd
/home/kt
$cd /home/kt/abc
$pwd
/home/kt/abc
```

Module - 3

5. a. Explain with a neat diagram, how a process can be initiated and how it can be terminated. **(10 Marks)**

Ans. Process Creation and Process termination are used to create and terminate processes respectively. Details about these are given as follows –

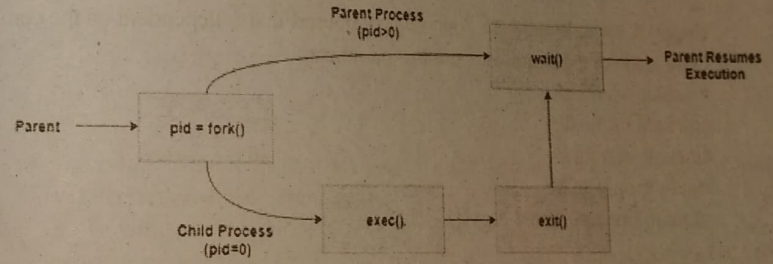
Process Creation

A process may be created in the system for different operations. Some of the events that lead to process creation are as follows –

- User request for process creation
- System Initialization
- Batch job initialization
- Execution of a process creation system call by a running process

A process may be created by another process using fork(). The creating process is called the parent process and the created process is the child process. A child process can have only one parent but a parent process may have many children. Both the parent and child processes have the same memory image, open files and environment strings. However, they have distinct address spaces.

A diagram that demonstrates process creation using fork() is as follows –



Process Creation Using fork()

Process Termination

Process termination occurs when the process is terminated The exit() system call is used by most operating systems for process termination.

Some of the causes of process termination are as follows –

1) A process may be terminated after its execution is naturally completed. This process leaves the processor and releases all its resources.
2) A child process may be terminated if its parent process requests for its termination.
3) A process can be terminated if it tries to use a resource that it is not allowed to. For example - A process can be terminated for trying to write into a read only file.

Sunstar Exam Scanner

4) If an I/O failure occurs for a process, it can be terminated. For example - If a process requires the printer and it is not working, then the process will be terminated.
5) In most cases, if a parent process is terminated then its child processes are also terminated. This is done because the child process cannot exist without the parent process.
6) If a process requires more memory than is currently available in the system, then it is terminated because of memory scarcity.

b. Explain fcntl API. Give an example to demonstrate file locking using fcntl API. **(10 Marks)**

Ans. The fcntl() function performs various actions on open descriptors, such as obtaining or changing the attributes of a file or socket descriptor.

Syntax

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int descriptor, int command,...)
```

descriptor

(Input) The descriptor on which the control command is to be performed, such as having its attributes retrieved or changed.

command

(Input) The command that is to be performed on the descriptor.

...

(Input) A variable number of optional parameters that is dependent on the command. Only some of the commands use this parameter.

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
char* file = argv[1];
int fd;
struct flock lock;
printf ("opening %s\n", file);
/* Open a file descriptor to the file. */
fd = open (file, O_WRONLY);
printf ("locking\n");
/* Initialize the flock structure. */
memset (&lock, 0, sizeof(lock));
lock.l_type = F_WRLCK;
/* Place a write lock on the file. */
fcntl (fd, F_SETLKW, &lock);
```


Sunstar Exam Scanner


```
printf ("locked; hit Enter to unlock... ");
/* Wait for the user to hit Enter. */
getchar ();
printf ("unlocking\n");
/* Release the lock. */
lock.l_type = F_UNLCK;
fcntl (fd, F_SETLKW, &lock);
close (fd);
return 0;
}
```

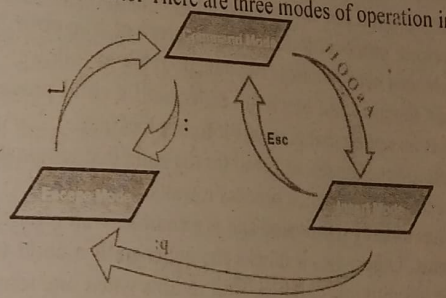
OR

6. a. What are the different modes of operations in vi editor? Explain with a suitable diagram. **(08 Marks)**

Ans. The default editor that comes with the UNIX operating system is called vi (visual editor). Using vi editor, we can edit an existing file or create a new file from scratch. we can also use this editor to just read a text file.

vi filename

Modes of Operation in vi editor There are three modes of operation in vi:



- **Command Mode:** When vi starts up, it is in Command Mode. This mode is where vi interprets any characters we type as commands and thus does not display them in the window. This mode allows us to move through a file, and to delete, copy, or paste a piece of text.
- To enter into Command Mode from any other mode, it requires pressing the [Esc] key. If we press [Esc] when we are already in Command Mode, then vi will beep or flash the screen.
- **Insert mode:** This mode enables you to insert text into the file. Everything that's typed in this mode is interpreted as input and finally, it is put in the file. The vi always starts in command mode. To enter text, you must be in insert mode. To come in insert mode you simply type i. To get out of insert mode, press the Esc key, which will put you back into command mode.
- **Last Line Mode(Escape Mode):** Line Mode is invoked by typing a colon [:], while vi is in Command Mode. The cursor will jump to the last line of the screen and vi will wait for a command. This mode enables you to perform tasks such as saving files, executing commands.

Sunstar Exam Scanner

b. Explain wait and waitpid APIs with their prototypes. Mention the differences between wait and waitpid. **(08 Marks)**

Ans. When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent. Because the termination of a child is an asynchronous event it can happen at any time while the parent is running this signal is the asynchronous notification from the kernel to the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler. The default action for this signal is to be ignored. We describe these options in Chapter 10. For now, we need to be aware that a process that calls wait or waitpid can

- Block, if all of its children are still running.
- Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched.
- Return immediately with an error, if it doesn't have any child processes.

If the process is calling wait because it received the SIGCHLD signal, we expect wait to return immediately. But if we call it at any random point in time, it can block.

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
/* Both return: process ID if OK, 0 (see later), or 1 on error */
```

The differences between these two functions are as follows.

- The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.
- The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, wait returns when one terminates. We can always tell which child terminated, because the process ID is returned by the function.

For both functions, the argument statloc is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument. If we don't care about the termination status, we simply pass a null pointer as this argument.

c. Explain the navigation keys for the following types of navigations in vi editor.
i) Movement in four direction
ii) Word navigation **(04 Marks)**

Ans. i) Movement in four direction:

Moving With Arrow Keys

If your machine is equipped with arrow keys, try these now. You should be able to move the cursor freely about the screen by using combinations of the up, down, right, and left arrow keys. Notice that you can only move the cursor across already existing text or input spaces.

If you're using vi from a remote terminal, the arrow keys might not work correctly.

Sunstar Exam Scanner

The arrow key behavior depends on your terminal emulator. If the arrow keys don't work for you, you can use the following substitutes:

- To move left, press h.
- To move right, press l.
- To move down, press j.
- To move up, press k.

ii) Word Navigation

Moving One Word

- Press w ("word") to move the cursor to the right one word at a time.
- Press b ("back") to move the cursor to the left one word at a time.
- Press W or B to move the cursor past the adjacent punctuation to the next or previous blank space.
- Press e ("end") to move the cursor to the last character of the current word.

Module - 4

7. a. Write a menu driven shell script to display list of files, process of users, today's date and users of the system. **(06 Marks)**

Ans.

```
clear
echo ----------------------------------------
echo '\tMenu Implementation'
echo ----------------------------------------
echo 1.Today DATE
echo 2.Process of the system
echo 3.Users of the system
echo 4.List of files
echo Enter your choice
read choice
case $choice in
    1)date;;
    2)ps;;
    3)who;;
    4)ls -l;;
    *)echo This is not a choice
Esac
```

Output:

```
----------------------------------------
Menu Implementation
----------------------------------------
1.Today DATE
2.Process of the system
3.Users of the system
4.List of files
Enter your choice
2
```

Sunstar Exam Scanner

```
PID TTY      TIME CMD
2458 pts/0   00:00:00 bash
2477 pts/0   00:00:00 sh
2479 pts/0   00:00:00 ps
```

b. Explain fork and vfork APIs with their prototypes and examples. (10 Marks)

Ans. An existing process can create a new one by calling the fork function

```
#include <unistd.h>
pid_t fork(void);
/* Returns: 0 in child, process ID of child in parent, 1 on error */
```

The new process created by fork is called the child process. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children. The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

```
#include "apue.h"
int glob = 6; /* external variable in initialized data */
char buf[] = "a write to stdout\n";
int
main(void)
{
  int var; /* automatic variable on the stack */
  pid_t pid;
  var = 88;
  if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
  printf("before fork\n"); /* we don't flush stdout */
  if ((pid = fork()) < 0) {
        err_sys("fork error");
  } else if (pid == 0) { /* child */
        glob++; /* modify variables */
        var++;
  } else {
        sleep(2); /* parent */
  }
  printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
  exit(0);
}
```

The function vfork has the same calling sequence and same return values as fork. But the semantics of the two functions differ. The vfork function originated with

Sunstar Exam Scanner

2.9BSD. Some consider the function a blemish, but all the platforms covered in this book support it. In fact, the BSD developers removed it from the 4.4BSD release, but all the open source BSD distributions that derive from 4.4BSD added support for it back into their own releases. The vfork function is marked as an obsolete interface in Version 3 of the Single UNIX Specification.

```
#include "apue.h"
int glob = 6; /* external variable in initialized data */
int main(void)
{
  int var; /* automatic variable on the stack */
  pid_t pid;
  var = 88;
  printf("before vfork\n"); /* we don't flush stdio */
  if ((pid = vfork()) < 0) {
        err_sys("vfork error");
  } else if (pid == 0) { /* child */
        glob++; /* modify parent's variables */
        var++;
  _exit(0); /* child terminates */
  }
  /*
  * Parent continues here.
  */
  printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
  exit(0);
}
```

c. What is FIFO? (04 Marks)

Ans. FIFOs are sometimes called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe.

```
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
/* Returns: 0 if OK, 1 on error */
```

OR

8. a. What are pipes? What are its limitations? Write a program to send data from parent to child over a pipe. (08 Marks)

Ans. Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations.

1) Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
2) Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

Sunstar Exam Scanner

A pipe is created by calling the pipe function.

```
#include <unistd.h>
int pipe(int filedes[2]);

/* Returns: 0 if OK, 1 on error */

#include "apue.h"
int main(void)
{
  int n;
  int fd[2];
  pid_t pid;
  char line[MAXLINE];
  if (pipe(fd) < 0)
        err_sys("pipe error");
  if ((pid = fork()) < 0) {
        err_sys("fork error");
  } else if (pid > 0) { /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
  } else { /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
  }
  exit(0);
}
```

b. Write a C/C++ program to Print where various types of data are stored. **(08 Marks)**

Ans.

```
#include "apue.h"
#include <sys/shm.h>
#define ARRAY_SIZE 40000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE 0600 /* user read/write */
char array[ARRAY_SIZE]; /* uninitialized data = bss */
int main(void)
{
  int shmid;
  char *ptr, *shmptr;
  printf("array[] from %lx to %lx\n", (unsigned long)&array[0],
        (unsigned long)&array[ARRAY_SIZE]);
  printf("stack around %lx\n", (unsigned long)&shmid);
```


Sunstar Exam Scanner


```
  if ((ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("malloc error");
  printf("malloced from %lx to %lx\n", (unsigned long)ptr,
        (unsigned long)ptr+MALLOC_SIZE);
  if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
        err_sys("shmget error");
  if ((shmptr = shmat(shmid, 0, 0)) == (void *)-1)
        err_sys("shmat error");
  printf("shared memory attached from %lx to %lx\n",
        (unsigned long)shmptr, (unsigned long)shmptr+SHM_SIZE);
  if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");
  exit(0);
}
```

c. Explain the differences between hard link and soft links? **(04 Marks)**

Ans.

COMPARISON PARAMETERS	HARD LINK	SOFT LINK
Inode number*	Files that are hard linked take the same inode number.	Files that are soft linked take a different inode number.
Directories	Hard links are not allowed for directories. (Only a superuser* can do it)	Soft links can be used for linking directories.
File system	It cannot be used across file systems.	It can be used across file systems.
Data	Data present in the original file will still be available in the hard links.	Soft links only point to the file name, it does not retain data of the file.
Original file's deletion	If the original file is removed, the link will still work as it accesses the data the original was having access to.	If the original file is removed, the link will not work as it doesn't access the original file's data.
Speed	Hard links are comparatively faster.	Soft links are comparatively slower.

Module - 5

9. a. What are daemon processes? Explain with a neat diagram the error logging facilities for a daemon process. **(10 Marks)**

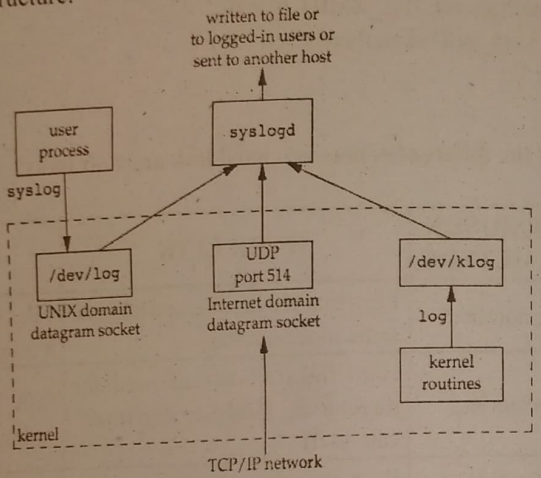
Ans. Daemons are processes that are often started when the system is bootstrapped and terminate only when the system is shut down. Because they don't have a controlling terminal, they run in the background. UNIX systems have numerous daemons that perform day-to-day activities.

Sunstar Exam Scanner

One problem a daemon has is how to handle error messages. It cannot (simply) write to:

- Standard error: it shouldn't have a controlling terminal.
- Console device: on many workstations the console device runs a windowing system.
- Separate files: it's a headache to keep up which daemon writes to which log file and to check these files on a regular basis.

A central daemon error-logging facility is required. The BSD syslog facility has been widely used since 4.2BSD. Most daemons use this facility. The following figure illustrates its structure:



There are three ways to generate log messages:

1) Kernel routines can call the log function. These messages can be read by any user process that opens and reads the /dev/klog device.
2) Most user processes (daemons) call the syslog(3) function to generate log messages. This causes the message to be sent to the UNIX domain datagram socket /dev/log.
3) A user process on this host or some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the syslog function never generates these UDP datagrams: they require explicit network programming by the process generating the log message.

The syslogd daemon reads all three forms of log messages. On start-up, this daemon reads a configuration file, usually /etc/syslog.conf, which determines where different classes of messages are to be sent. For example, urgent messages can be sent to the system administrator (if logged in) and printed on the console, whereas warnings may be logged to a file.

```
#include <syslog.h>
void openlog(const char *ident, int option, int facility);
void syslog(int priority, const char *format, ...);
void closelog(void);
```



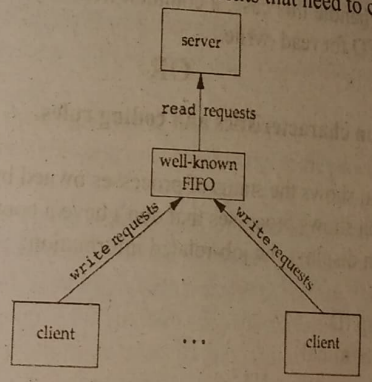
```
int setlogmask(int maskpri);
/* Returns: previous log priority mask value */
```

- Calling openlog is optional. If it's not called, the first time syslog is called, openlog is called automatically.
- Calling closelog is also optional. It closes the descriptor being used to communicate with the syslogd daemon.
- The openlog function:

1) indent argument is the name of the program.
2) option argument is a bitmask specifying various options. (see the table below)
3) facility argument lets the configuration file specify that messages from different facilities are to be handled differently. If we don't call openlog, or if we call it with a facility of 0, we can still specify the facility as part of the priority argument to syslog.

b. With a neat diagram explain client server communication using FIFO **(10 Marks)**

Ans. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. "well-known" means that the pathname of the FIFO is known to all the clients that need to contact the server.

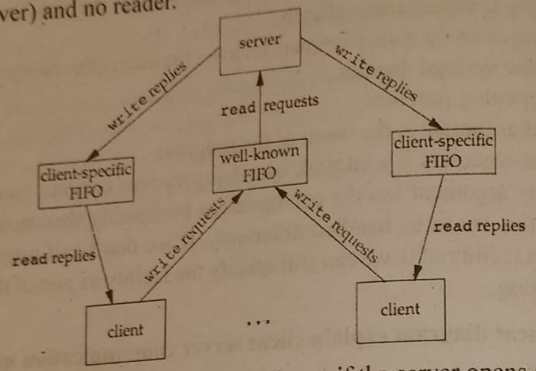


Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size. This prevents any interleaving of the client writes.

The problem in using FIFOs for this type of client–server communication is how to send replies back from the server to each client. A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client's process ID. For example, the server can create a FIFO with the name /tmp/serv1.XXXXX, where XXXXX is replaced with the client's process ID. This arrangement is shown the figure below.

This arrangement works, although it is impossible for the server to tell whether a client crashes. A client crash leaves the client-specific FIFO in the file system. The

server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.



With the arrangement shown in the figure above, if the server opens its well-known FIFO read-only (since it only reads from it) each time the number of clients goes from 1 to 0, the server will read an end of file on the FIFO. To prevent the server from having to handle this case, a common trick is just to have the server open its well-known FIFO for read–write.

OR

10. a. Explain daemon characteristics and coding rules. **(10 Marks)**

Ans. Characteristics

- The -a option shows the status of processes owned by others.
- The -x option shows processes that don't have a controlling terminal.
- The -j option displays the job-related information:

1) Session ID
2) Process group ID
3) Controlling terminal
4) Terminal process group ID

Coding Rules

1) Call umask to set the file mode creation mask to a known value, usually 0.

- If the daemon process creates files, it may want to set specific permissions.
- On the other hand, if the daemon calls library functions that result in files being created, then it might make sense to set the file mode create mask to a more restrictive value (such as 007), since the library functions might not allow the caller to specify the permissions through an explicit argument.

2) Call fork and have the parent exit. This does several things:

- If the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done
- The child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader. This is a prerequisite for the call to setsid that is done next.

Sunstar Exam Scanner

3) Call setsid to create a new session. The process:

- becomes the leader of a new session,
- becomes the leader of a new process group,
- and is disassociated from its controlling terminal.

4) Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.
Alternatively, some daemons might change the current working directory to a specific location where they will do all their work. For example, a line printer spooling daemon might change its working directory to its spool directory.
5) Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent (which could be a shell or some other process). We can use our open_max function or the getrlimit function to determine the highest descriptor and close all descriptors up to that value.
6) Some daemons open file descriptors 0, 1, and 2 to /dev/null so that any library routines that try to read from standard input or write to standard output or standard error will have no effect.

b. Briefly explain the Kill() API and alarm() API. **(10 Marks)**

Ans. **i) Kill():** The kill() system call can be used to send any signal to any process group or process.

- If pid is positive, then signal sig is sent to pid.
- If pid equals 0, then sig is sent to every process in the process group of the current process.
- If pid equals -1, then sig is sent to every process for which the calling process has permission to send signals, except for process 1 (init), but see below.
- If pid is less than -1, then sig is sent to every process in the process group -pid.
- If sig is 0, then no signal is sent, but error checking is still performed.

For a process to have permission to send a signal it must either be privileged (under Linux: have the CAP_KILL capability), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of SIGCONT it suffices when the sending and receiving processes belong to the same session.

RETURN VALUE

On success (at least one signal was sent), zero is returned. On error, -1 is returned, and errno is set appropriately.

ERRORS

Tag	Description
EINVAL	An invalid signal was specified.
EPERM	The process does not have permission to send the signal to any of the target processes.
ESRCH	The pid or process group does not exist. Note that an existing process might be a zombie process which already committed termination, but has not yet been wait()ed for.

Syntax:

Sunstar Exam Scanner

#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);

ii) Alarm() alarm() arranges for a SIGALRM signal to be delivered to the process in seconds.

- If seconds is zero, no new alarm() is scheduled.
- In any event any previously set alarm() is cancelled.

RETURN VALUE

alarm() returns the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.

alarm() and setitimer(2) share the same timer; calls to one will interfere with use of the other.

Alarms created by alarm() are preserved across execve(2) and are not inherited by children created via fork(2).

sleep(3) may be implemented using SIGALRM; mixing calls to alarm() and sleep(3) is a bad idea.

Scheduling delays can, as ever, cause the execution of the process to be delayed by an arbitrary amount of time.

Syntax:

#include <unistd.h>
unsigned int alarm(unsigned int seconds);

Fifth Semester B.E. Degree Examination
CBCS - Model Question Paper - 2
UNIX PROGRAMMING

Time: 3 hrs. Max. Marks: 100

Note : Answer any FIVE full questions, selecting ONE full question from each module.

Module - 1

1. a. **Explain the following commands with example.**
i) who ii) echo iii) date **(08 Marks)**

Ans. **i) who command** is used to find out the following information:
1. Time of last system boot
2. Current run level of the system
3. List of logged in users and more.

The who command is used to get information about currently logged in user on to system.

Syntax: $who [options] [filename]

Examples:

The who command displays the following information for each user currently logged in to the system if no option is provided:

Login name of the users
Terminal line numbers
Login time of the users in to system
Remote host name of the user

niranjankumar console Sep 24 14:46
niranjankumar ttys000 Sep 29 18:41

ii) echo command in linux is used to display line of text/string that are passed as an argument. This is a built-in command that is mostly used in shell scripts and batch files to output status text to the screen or a file.

Syntax:
echo [option] [string]

Displaying a text/string:
Syntax: echo [string]

Niranjans-Air:a niranjankumar$ echo "Unix Programming"
Unix Programming

iii) date command is used to display the system date and time. date command is also used to set date and time of the system. By default the date command displays the date in the time zone on which unix/linux operating system is configured.You must be the super-user (root) to change the date and time.

Syntax:
date [OPTION]... [+FORMAT]
date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]

date (no option) : With no options, the date command displays the current date and time, including the abbreviated day name, abbreviated month name, day of the month, the time separated by colons, the time zone name, and the year.

Command:

$date

Output:

Sun Oct 11 22:46:01 IST 2020

b. what are major differences between ANSI C and K and R C? Explain with example. (10 Marks)

Ans. Following are the major differences between ANSI C and K&R C (Kernighan and Ritchie):

Support function prototyping.

Support const and volatile data type qualifiers.

Support wide characters and internationalization.

Permit function pointers to be used without dereferencing.

1. Function Prototyping:

Function prototype includes function names, arguments, data types, and return value data types.

In ANSI C when we create any function with an argument if we want to call that function then we have to give the same arguments and same data type same applies to return value too.

Example:

```
//creating a function in ANSI C
unsigned long abc(char *a, double b)
{
/** body/content**/
}
//calling function in ANSI C
abc(char *a, double b);
```

"It fixes the major issue of K&R C because in K&R C we can create or call a function without an argument which leads to program crash."

2. Support const and volatile data type qualifiers.

If we declare some data using constant keyword than that data become fixed we can't change its value later.

Example:

```
int printf(const char* abc.....);
```

Declares a abc argument that is of a const char * data type, meaning that the function printf cannot modify data in any character array that is passed as an actual argument value to abc.

Volatile keyword specifies that the values of some variables may change asynchronously, giving a hint to the compiler's optimization algorithm not to remove any "redundant" statements that involve "volatile" objects.

eg:

```
char get_io()
{
volatile char* io_port = 0x7777;
char ch = *io_port; /*read first byte of data*/
ch = *io_port; /*read second byte of data*/
}
```

If io_port variable is not declared to be volatile when the program is compiled, the compiler may eliminate second ch = *io_port statement, as it is considered redundant with respect to the previous statement.

"K&RC didn't support const and volatile data type qualifiers"

3. Wide characters and internationalization.

ANSI C allows a wide character (more than one byte of storage per character). setlocale function is defined in ANSI C, which allows users to specify the format of the date, monetary and real number representations.

Function prototype of setlocale function:

```
#include<locale.h>
char setlocale (int category, const char* locale);
```

"K&RC didn't support wide character and internationalization"

4. Function pointers without dereferencing

ANSI C specifies that a function pointer may be used like a function name. No referencing is needed when calling a function whose address is contained in the pointer.

For Example, the following statement given below defines a function pointer funptr, which contains the address of the function foo.

```
extern void foo(double xyz,const int *ptr);
void (*funptr)(double,const int *)=foo;
```

The function foo may be invoked by either directly calling foo or via the funptr.

```
foo(12.78,"Hello world");
funptr(12.78,"Hello world");
```

K&R C requires funptr be dereferenced to call foo. (* funptr) (13.48,"Hello usp");

ANSI C also defines a set of C processor(cpp) symbols, which may be used in user programs. These symbols are assigned actual values at compilation time.

ANSI C is an Upgraded version of K&R C

c. Differentiate the commands echo "$SHELL" and echo '$SHELL' (02 Marks)

Ans. echo "$SHELL" will print the path value stored in the variable SHELL

echo '$SHELL' will print $SHELL on the terminal, since text inside single quote treated as string.

OR

2. a. write a program to check the following limits using function defined by POSIX-1.
i) Number of clock ticks per seconds
ii) Maximum number of real time signals
iii) Maximum number of links a file may have
iv) Number of simultaneous asynchronous input/output
v) Maximum length in bytes of a pathname. **(10 Marks)**

Ans.

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include<iostream>
#include<unistd.h>
int main()
{
 using namespace std;
 int res;
 if((res=sysconf(_SC_CLK_TCK))==-1)
        cout<<"System doesnot support\n";
 else
        cout<<"Number of Clock Tick:"<<res<<endl;
 if((res=sysconf(_SC_CHILD_MAX))==-1)
        cout<<"System does not support\n";
 else
        cout<<"Maximum Number of Child Process that process can
create:"<<res<<endl;
 if((res=pathconf("/", _PC_PATH_MAX))==-1)
        cout<<"System does not support\n";
 else
        cout<<"Maximum Path Length:"<<res<<endl;
 if((res=pathconf("/", _PC_NAME_MAX))==-1)
        cout<<"System does not support\n";
 else
        cout<<"Maximum No.of Character in a filename:"<<res<<endl;
 if((res=sysconf(_SC_OPEN_MAX))==-1)
        cout<<"System does not support\n";
 else
        cout<<"Maximum Number of opened files perprocess:"<<res<<endl;
 return 0;
}
```

b. Explain with example the date command with all options **(10 Marks)**

Ans. date command is used to display the system date and time. date command is also used to set date and time of the system. By default the date command displays the

Sunstar Exam Scanner

date in the time zone on which unix/linux operating system is configured.You must be the super-user (root) to change the date and time.

Syntax:

date [OPTION]... [+FORMAT]

date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]

Options with Examples

1. date (no option) : With no options, the date command displays the current date and time, including the abbreviated day name, abbreviated month name, day of the month, the time separated by colons, the time zone name, and the year.

Command:

$date

Output:

Sun Oct 11 23:43:47 IST 2020

Note : Here unix system is configured in pacific daylight time.

2. -u Option: Displays the time in GMT (Greenwich Mean Time)/ UTC (Coordinated Universal Time) time zone.

Command:

$date -u

Output :

Sun Oct 11 18:14:25 UTC 2020

3. -date or -d Option: Displays the given date string in the format of date. But this will not affect the system's actual date and time value. Rather it uses the date and time given in the form of string.

Syntax:

$date --date=" string "

Command:

$date --date="2/02/2010"

$date --date="Feb 2 2010"

Output:

Tue Feb 2 00:00:00 PST 2010

Tue Feb 2 00:00:00 PST 2010

4. Using –date option for displaying past dates:

- Date and time of 2 years ago.

Command:

$date --date="2 year ago"

Output:

Sat Oct 10 23:42:15 PDT 2015

- Date and time of 5 seconds ago.

Command:

$date --date="5 sec ago"

Output:

Tue Oct 10 23:45:02 PDT 2017

Sunstar Exam Scanner

- Date and time of previous day.

Command:

$date --date="yesterday"

Output:

Mon Oct 9 23:48:00 PDT 2017

- Date and time of 2 months ago.

Command:

$date --date="2 month ago"

Output:

Thu Aug 10 23:54:51 PDT 2017

- Date and time of 10 days ago.

Command:

$date --date="10 day ago"

Output:

Sat Sep 30 23:56:55 PDT 2017

5. Using –date option for displaying future date:

- Date and time of upcoming particular week day.

Command:

$date --date="next tue"

Output:

Tue Oct 17 00:00:00 PDT 2017

- Date and time after two days.

Command:

$date --date="2 day"

Output:

Fri Oct 13 00:05:52 PDT 2017

- Date and time of next day.

Command:

$date --date="tomorrow"

Output:

Thu Oct 12 00:08:47 PDT 2017

- Date and time after 1 year on the current day.

Command:

$date --date="1 year"

Output:

Thu Oct 11 00:11:38 PDT 2018

6. s or –set Option: To set the system date and time -s or –set option is used.

Syntax:

$date --set="date to be set"

Command:

$date

Output:

Wed Oct 11 15:23:26 PDT 2017

Sunstar Exam Scanner

Command:

$date --set="Tue Nov 13 15:23:34 PDT 2018"

$date

Output:

Tue Nov 13 15:23:34 PDT 2018

7. file or -f Option: This is used to display the date string present at each line of file in the date and time format. This option is similar to –date option but the only difference is that in –date we can only give one date string but in a file we can give multiple date strings at each line.

Syntax:

$date --file=file.txt

$cat >> datefile

Sep 23 2018

Nov 03 2019

Command:

$date --file=datefile

Output:

Sun Sep 23 00:00:00 PDT 2018

Sun Nov 3 00:00:00 PDT 2019

8. r Option: This is used to display the last modified timestamp of a datefile .

Syntax:

$date -r file.txt

We can modify the timestamp of a datefile by using touch command.

$touch datefile

$date

Wed Oct 11 15:54:18 PDT 2017

//this is the current date and time

$touch datefile

//The timestamp of datefile is changed using touch command.

This was done few seconds after the above date command's output.

$date

Wed Oct 11 15:56:23 PDT 2017

//display last modified time of datefile

9. List of Format specifiers used with date command:

%D: Display date as mm/dd/yy.

%d: Display the day of the month (01 to 31).

%a: Displays the abbreviated name for weekdays (Sun to Sat).

%A: Displays full weekdays (Sunday to Saturday).

%h: Displays abbreviated month name (Jan to Dec).

%b: Displays abbreviated month name (Jan to Dec).

%B: Displays full month name(January to December).

%m: Displays the month of year (01 to 12).

%y: Displays last two digits of the year(00 to 99).

Sunstar Exam Scanner

%Y: Display four-digit year.
%T: Display the time in 24 hour format as HH:MM:SS.
%H: Display the hour.
%M: Display the minute.
%S: Display the seconds.

Syntax:
$date +%[format-option]
Examples:
Command:
$date "+%D"
Output:
10/11/17
Command:
$date "+%D %T"
Output:
10/11/17 16:13:27
Command:
$date "+%Y-%m-%d"
Output:
2017-10-11
Command:
$date "+%Y/%m/%d"
Output:
2017/10/11
Command:
$date "+%A %B %d %T %y"
Output:
Thursday October 07:54:29 17

Module - 2

3. a. Explain chmod and fchmod function prototype, list and explain all the modes (10 Marks)

Ans. These two functions allow us to change the file access permissions for an existing file.

```
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
int fchmod(int filedes, mode_t mode);
/* Both return: 0 if OK, 1 on error. */
```

The chmod function operates on the specified file, whereas the fchmod function operates on a file that
has already been opened.
To change the permission bits of a file, the effective user ID of the process must be equal to the

Sunstar Exam Scanner

owner ID of the file, or the process must have superuser permissions.

Mode	Description
S_ISUID	set-user-ID on execution
S_ISGID	set-group-ID on execution
S_ISVTX	saved-text (sticky bit)
S_IRWXU	read, write, and execute by user (owner)
S_IRUSR	read by user (owner)
S_IWUSR	write by user (owner)
S_IXUSR	execute by user (owner)
S_IRWXG	read, write, and execute by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXO	read, write, and execute by other
S_IROTH	read by other (world)
S_IWOTH	write by other (world)
S_IXOTH	execute by other (world)

b. Explain if condition in shell scripting, with example. (06 marks)

Ans. The if...elif...fi statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

If command is successful then execute commands else execute commands fi	If command is successful then execute commands fi	If command is successful then execute commands elif commands is successful then else.... fi

```
#!/bin/sh
# emp3.sh: Using if and else
#
if grep "^$1" /etc/passwd 2>/dev/null   # Search username at beginning of line
then
        echo "Pattern found - Job Over"
else
        echo "Pattern not found"
fi
```

Sunstar Exam Scanner

c. Write a shell program to find factorial of a number using while loop. **(04 Marks)**

Ans.
```
#shell script for factorial of a number
#factorial using while loop
echo "Enter a number"
read num
fact=1
while [ $num -gt 1 ]
do
 fact=$((fact * num))  #fact = fact * num
 num=$((num - 1))      #num = num - 1
done
echo $fact
```

OR

4. a. Explain grep and egrep with examples. **(12 Marks)**

Ans. The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern. The pattern that is searched in the file is referred to as the regular expression (grep stands for globally search for regular expression and print out).

Syntax:

grep [options] pattern [files]

Options Description

-c : This prints only a count of the lines that match a pattern
-h : Display the matched lines, but do not display the filenames.
-i : Ignores, case for matching
-l : Displays list of a filenames only.
-n : Display the matched lines and their line numbers.
-v : This prints out all the lines that do not matches the pattern
-e exp : Specifies expression with this option. Can use multiple times.
-f file : Takes patterns from file, one per line.
-E : Treats pattern as an extended regular expression (ERE)
-w : Match whole word
-o : Print only the matched parts of a matching line,
with each such part on a separate output line.

Sample Commands

Consider the below file as an input.

$cat > geekfile.txt

unix is great os. unix is opensource. unix is free os. learn operating system.
Unix linux which one you choose. uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

1. Case insensitive search : The -i option enables to search for a string case insensitively in the give file. It matches the words like "UNIX", "Unix", "unix".

Sunstar Exam Scanner

$grep -i "UNix" geekfile.txt

Output:

unix is great os. unix is opensource. unix is free os.
Unix linux which one you choose.
uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

2. Displaying the count of number of matches: We can find the number of lines that matches the given string/pattern

$grep -c "unix" geekfile.txt

Output:

2

egrep is a pattern searching command which belongs to the family of grep functions. It works the same way as grep -E does. It treats the pattern as an extended regular expression and prints out the lines that match the pattern. If there are several files with the matching pattern, it also displays the file names for each line.

Syntax:

egrep [options] 'PATTERN' files

Options: Most of the options for this command are same as grep.

-c: Used to counts and prints the number of lines that matched the pattern and not the lines.

```
5
anindo@anindo-One-Z1402:~\Documents$ egrep -c Hello hello.c
```

-v: It prints the lines that does not match with the pattern.

```
anindo@anindo-One-Z1402:~/Documents$ egrep -v Hello hello.c
#include "stdio.h"

int main(){
return 0;
}
```

-i: Ignore the case of the pattern while matching.

```
anindo@anindo-One-Z1402:~/Documents$ egrep -i hello hello.c
printf("Hello World!");
printf("-Hello!");
```

-l: Prints only the names of the files that matched. It does not mention the matching line numbers or any other information.

```
anindo@anindo-One-Z1402:~/Documents$ egrep -l Hello hello.c
```

-L: Prints only the names of the files that did not have the pattern. Opposite of -l flag.

```
anindo@anindo-One-Z1402:~/Documents$ egrep -L Hello hello.c myfile.txt
```

-e: Allows to use a '-' sign in the beginning of the pattern. If not mentioned the shell tries to execute the pattern as an option and returns an error.

Sunstar Exam Scanner

```
anindo@anindo-One-Z1402:~/Documents$ egrep -e -Hello hello.c
printf("-Hello!");
```

-w: Prints only those lines that contain the whole words. Word-constituent characters are letters, digits and underscore. The matching substring must be seperated by non-word constituent characters.

```
anindo@anindo-One-Z1402:~/Documents$ egrep -w Hello hello.c
printf("Hello World!");
printf("-Hello!");
```

-x: Prints only those lines that matches an entire line of the file.

```
anindo@anindo-One-Z1402:~/Documents$ egrep -x 'return 0;' hello.c
return 0;
```

-m NUMBER: Continue to search for matches till the count reaches NUMBER mentioned as argument.

```
anindo@anindo-One-Z1402:~/Documents$ egrep -m 1 Hello hello.c
printf("Hello World!");
```

-o: Prints only the matched parts of the line and not the entire line for each match.

```
anindo@anindo-One-Z1402:~/Documents$ egrep -o Hello hello.c
Hello
Hello
```

-n: Prints each matched line along with the respective line numbers. For multiple files, prints the file names along with line numbers.

```
anindo@anindo-One-Z1402:~/Documents$ egrep -n Hello hello.c
4:printf("Hello World!");
5:printf("-Hello!");
```

-r: Recursively search for the pattern in all the files of the directory. The last argument is the directory to check. '.' (dot) represents the current directory.

```
anindo@anindo-One-Z1402:~/Documents$ egrep -r -i 'h*' .
:hi
:#include "stdio.h"
:int main(){
:printf("Hello World!");
:printf("-Hello!");
:return 0;
:}
```


b. Briefly explain Basic Regular Expression (BRE) and Extended Regular Expression (ERE) metacharacters. (10 Marks)

Ans. The Basic Regular Expressions or BRE flavor standardizes a flavor similar to the one used by the traditional UNIX grep command. This is pretty much the oldest regular expression flavor still in use today. One thing that sets this flavor apart is that most metacharacters require a backslash to give the metacharacter its flavor. Most other flavors, including POSIX ERE, use a backslash to suppress the meaning of metacharacters. Using a backslash to escape a character that is never a metacharacter is an error.

A BRE supports POSIX bracket expressions, which are similar to character classes in other regex flavors, with a few special features. Shorthands are not supported. Other features using the usual metacharacters are the dot to match any character except a line break, the caret and dollar to match the start and end of the string, and the star to repeat the token zero or more times. To match any of these characters literally, escape them with a backslash.

The other BRE metacharacters require a backslash to give them their special meaning. The reason is that the oldest versions of UNIX grep did not support these. The developers of grep wanted to keep it compatible with existing regular expressions, which may use these characters as literal characters. The BRE a{1,2} matches a{1,2} literally, while a\{1,2\} matches a or aa. Some implementations support \? and \+ as an alternative syntax to \{0,1\} and \{1,\}, but \? and \+ are not part of the POSIX standard. Tokens can be grouped with \(and \). Backreferences are the usual \1 through \9. Only up to 9 groups are permitted. E.g. \(ab\)\1 matches abab, while (ab)\1 is invalid since there's no capturing group corresponding to the backreference \1. Use \\1 to match \1 literally.

Desired pattern	Basic (BRE) Syntax	Extended (ERE) Syntax

literal '+' (plus sign)	$ echo 'a+b=c' > foo
	$ sed -n '/a+b/p' foo
	a+b=c
		$ echo 'a+b=c' > foo
		$ sed -E -n '/a\+b/p' foo
		a+b=c

The Extended Regular Expressions or ERE flavor standardizes a flavor similar to the one used by the UNIX egrep command. "Extended" is relative to the original UNIX grep, which only had bracket expressions, dot, caret, dollar and star. An ERE support these just like a BRE. Most modern regex flavors are extensions of the ERE flavor. By today's standard, the POSIX ERE flavor is rather bare bones. The POSIX standard was defined in 1986, and regular expressions have come a long way since then.

The developers of egrep did not try to maintain compatibility with grep, creating a separate tool instead. Thus egrep, and POSIX ERE, add additional metacharacters without backslashes. You can use backslashes to suppress the meaning of all metacharacters, just like in modern regex flavors. Escaping a character that is not a metacharacter is an error.

The quantifiers ?, +, {n}, {n,m} and {n,} repeat the preceding token zero or once, once or more, n times, between n and m times, and n or more times, respectively.

Alternation is supported through the usual vertical bar |. Unadorned parentheses create a group, e.g. (abc){2} matches abcabc. The POSIX standard does not define backreferences. Some implementations do support \1 through \9, but these are not part of the standard for ERE. ERE is an extension of the old UNIX grep, not of POSIX BRE.

Desired pattern	Basic (BRE) Syntax	Extended (ERE) Syntax

One or more 'a' characters followed by 'b' (plus sign as special meta-character)	$ echo aab > foo
	$ sed -n '/a\+b/p' foo
	aab
		$ echo aab > foo
		$ sed -E -n '/a+b/p' foo
		aab

Module - 3

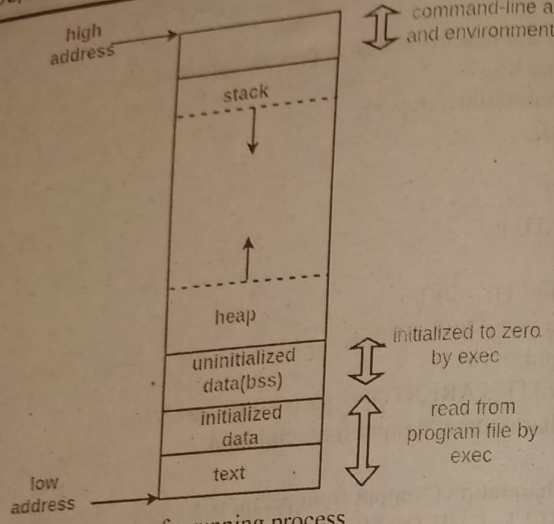
5. a. What is race condition in process control? Write a program to demonstrate race condition and resolve the same. **(10 Marks)**

Ans. A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run. The fork function is a lively breeding ground for race conditions, if any of the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork. In general, we cannot predict which process runs first. Even if we knew which process would run first, what happens after that process starts running depends on the system load and the kernel's scheduling algorithm.

Program with a race condition

```
#include "apue.h"
static void charatatime(char *);
int main(void)
{
 pid_t pid;
 if ((pid = fork()) < 0) {
        err_sys("fork error");
 } else if (pid == 0) {
        charatatime("output from child\n");
 } else {
        charatatime("output from parent\n");
 }
        exit(0);
}
static void charatatime(char *str)
{
char *ptr;
int c;
setbuf(stdout, NULL); /* set unbuffered */
for (ptr = str; (c = *ptr++) != 0; )
```


Sunstar Exam Scanner



A typical memory layout of a running process

1. Text Segment: A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

2. Initialized Data Segment:

Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

For instance the global string defined by char s[] = "hello world" in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const char* string = "hello world" makes the string literal "hello world" to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.

Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

Sunstar Exam Scanner

3. Uninitialized Data Segment:

Uninitialized data segment, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared static int i; would be contained in the BSS segment.

For instance a global variable declared int j; would be contained in the BSS segment.

4. Stack:

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

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A "stack pointer" register tracks the top of the stack; it is adjusted each time a value is "pushed" onto the stack. The set of values pushed for one function call is termed a "stack frame"; A stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.

5. Heap:

Heap is the segment where dynamic memory allocation usually takes place.

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

OR

6. a. What are setjmp() and longjmp() Functions in Standard C Library. Explain with Examples When to Use them. **(10 Marks)**

Ans. 'setjmp()' and 'longjmp()' functions provide a powerful mechanism similar to 'goto' but not restricted to just current function. This mechanism is useful in situations

Sunstar Exam Scanner

where there's a deeply nested function calls and when error gets detected, in some lower-level function, then control immediately transfers to top-level function without having to return and check for error code by intermediate functions. These functions work in pairs as follows:

1. setjmp(jmp_buf resume_here) must be called first. This initializes 'resume_here' with program state information viz. value of pointer to the top of the stack, value of program counter and some other info. It returns ZERO when returns first time.
2. longjmp(jmp_buf resume_here, int i) is then called. This says go back to place where to resume from defined by 'resume_here'. This makes it look like you're returning from original 'setjmp()' but return value of integer 'i' to let code tell when you actually got back here via 'longjmp()'.
3. After longjmp()'ed, contents of 'jmp_buf' variable 'resume_here' destroys.

Let's take a C program to learn how to use them.

```
/* setjmp_longjmp.c -- program handles error through 'setjmp()' */
/* and longjmp() */
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

/* declare variable of type jmp_buf */
jmp_buf resume_here;

void hello(void);

int main(void)
{
  int ret_val;

  /* Initialize 'resume_here' by calling setjmp() */
  if (setjmp(resume_here)) {
    printf("After \'longjump()\', back in \'main()\'\n");
    printf("\'jump buffer variable \'resume_here\'\' becomes "
          "INVALID!\n");
  }
  else {
    printf("\'setjmp()\' returns first time\n");
    hello();
  }

  return 0;
}

void hello(void)
```


Sunstar Exam Scanner


```
{
  printf("Hey, I'm in \'hello()\'\n");
  longjmp(resume_here, 1);

  /* other code */
  printf("can't be reached here because I did longjmp!\n");
}
```

Output is written below

```
'setjmp()' returns first time
Hey, I'm in 'hello()'
After 'longjump()', back in 'main()'
'jump buffer variable 'resume_here" becomes INVALID!
```

Notice that when 'setjmp()' called first time, returned value ZERO and hence 'else clause' executed. Within there, after printing the msg "'setjmp()' returns first time" called function 'hello()'. In 'hello()' function, 'longjmp()' called which caused control immediately transferred back to 'setjmp()' in the 'main()'. This time, 'setjmp()' returned value 1 passed to it by 'longjmp()' and so 'if clause' executed.

A 'goto' can't jump out of current function while we can 'longjmp' long way away, even to a function in a different file.

b. Differentiate between Hardlinks and Softlinks

Ans. **(04 Marks)**

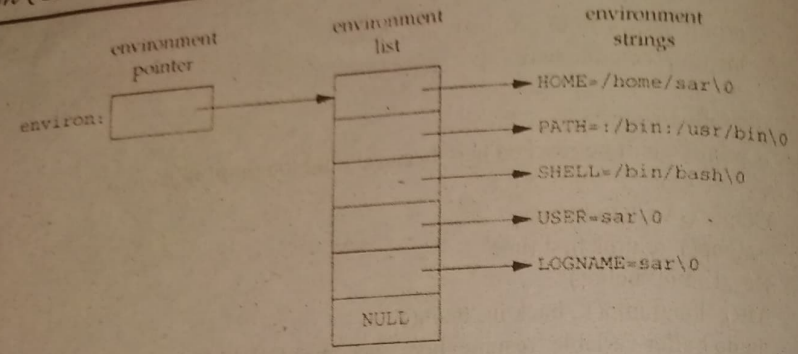
Hard Links	Soft link (symbolic link)
Target must exist	Target may already exist, but does not have to
Allowed within file systems only	Allowed between different file systems
Links directly to the place the file is stored	Links to the entry in the file system table (node)
Removing the link means removing the whole file	Removing the link means removing the link to the node, not the file itself

c. Write a note on Environment List **(06 Marks)**

Ans. Each program is also passed an environment list. Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string.

The address of the array of pointers is contained in the global variable environ: extern char **environ; For example, if the environment consisted of five strings, it could look like Figure below. Here we explicitly show the null bytes at the end of each string. We'll call environ the environment pointer, the array of pointers the environment list, and the strings they point to the environment strings.

Sunstar Exam Scanner



By convention, the environment consists of name=value strings, as shown in Figure. Most predefined names are entirely uppercase, but this is only a convention.

Module - 4

7. a. Explain Semaphores in detail. **(08 Marks)**

Ans. A semaphore isn't a form of IPC similar to the others that we've described (pipes, FIFOs, and message queues). A semaphore is a counter used to provide access to a shared data object for multiple processes.

The Single UNIX Specification includes an alternate set of semaphore interfaces in the semaphore option of its real-time extensions. We do not discuss these interfaces in this text. To obtain a shared resource, a process needs to do the following:

Test the semaphore that controls the resource.

If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.

Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.

To implement semaphores correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation. For this reason, semaphores are normally implemented inside the kernel.

A common form of semaphore is called a binary semaphore. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing. XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore,

Sunstar Exam Scanner

we specify the number of values in the set.

The creation of a semaphore (semget) is independent of its initialization (semctl). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.

Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated.

b. Explain the steps involved in execution of man command in order to change the userIDs. **(08 Marks)**

Ans. Assuming that the man program file is owned by the user name man and has its set-user-ID bit set, when we exec it, we have

real user ID = our user ID
effective user ID = man
saved set-user-ID = man

The man program accesses the required configuration files and manual pages. These files are owned by the user name man, but because the effective user ID is man, file access is allowed.

Before man runs any command on our behalf, it calls setuid(getuid()). Because we are not a superuser process, this changes only the effective user ID. We have

real user ID = our user ID (unchanged)
effective user ID = our user ID
saved set-user-ID = man (unchanged)

Now the man process is running with our user ID as its effective user ID. This means that we can access only the files to which we have normal access. We have no additional permissions. It can safely execute any filter on our behalf.

When the filter is done, man calls setuid(euid), where euid is the numerical user ID for the user name man. (This was saved by man by calling geteuid.) This call is allowed because the argument to setuid equals the saved set-user-ID. (This is why we need the saved set-user-ID.) Now we have

real user ID = our user ID (unchanged)
effective user ID = man
saved set-user-ID = man (unchanged)

The man program can now operate on its files, as its effective user ID is man.

c. Explain the popen and pclose prototypes. **(04 Marks)**

Ans. Since a common operation is to create a pipe to another process, to either read its output or send it input, the standard I/O library has historically provided the popen and pclose functions. These two functions handle all the dirty work that we've been doing ourselves: creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>
FILE *popen(const char *cmdstring, const char *type);
/* Returns: file pointer if OK, NULL on error */
```

Sunstar Exam Scanner

int pclose(FILE *fp);
/* Returns: termination status of cmdstring, or -1 on error */

OR

8. a. Explain the msqid_ds structure associated with queue. **(04 Marks)**

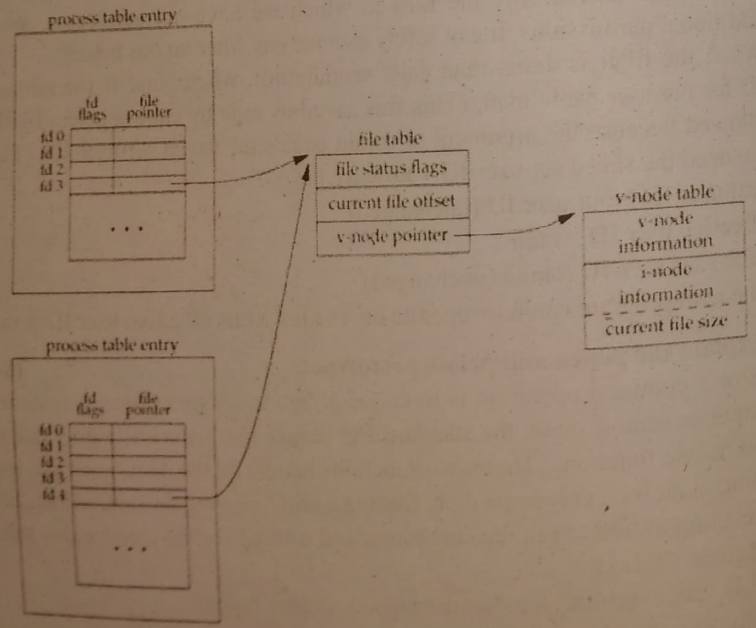
Ans. Each queue has the following msqid_ds structure associated with it:

```
struct msqid_ds {
struct ipc_perm msg_perm; /* see Section 15.6.2 */
msgqnum_t msg_qnum; /* # of messages on queue */
msglen_t msg_qbytes; /* max # of bytes on queue */
pid_t msg_lspid; /* pid of last msgsnd() */
pid_t msg_lrpid; /* pid of last msgrcv() */
time_t msg_stime; /* last-msgsnd() time */
time_t msg_rtime; /* last-msgrcv() time */
time_t msg_ctime; /* last-change time */
.
.
.
};
```

This structure defines the current status of the queue. The members shown are the ones defined by the Single UNIX Specification.

b. With neat diagram, elaborate passing an open file from the top process to the bottom process. **(10 Marks)**

Ans.



Sunstar Exam Scanner

The ability to pass an open file descriptor between processes is powerful. It can lead to different ways of designing clientserver applications. It allows one process (typically a server) to do everything that is required to open a file (involving such details as translating a network name to a network address, dialing a modem, negotiating locks for the file, etc.) and simply pass back to the calling process a descriptor that can be used with all the I/O functions. All the details involved in opening the file or device are hidden from the client.

When we pass an open file descriptor from one process to another, we want the passing process and the receiving process to share the same file table entry. Above Figure shows the desired arrangement.

Technically, we are passing a pointer to an open file table entry from one process to another. This pointer is assigned the first available descriptor in the receiving process. (Saying that we are passing an open descriptor mistakenly gives the impression that the descriptor number in the receiving process is the same as in the sending process, which usually isn't true.) Having two processes share an open file table is exactly what happens after a fork.

What normally happens when a descriptor is passed from one process to another is that the sending process, after passing the descriptor, then closes the descriptor. Closing the descriptor by the sender doesn't really close the file or device, since the descriptor is still considered open by the receiving process (even if the receiver hasn't specifically received the descriptor yet).

c. Explain message queue APIs with their prototypes. **(06 Marks)**

Ans. A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by msgget(). New messages are added to the end of a queue by msgsnd(). Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by msgrcv(). We don't have to fetch the messages in a first-in, first out order. Instead, we can fetch messages based on their type field.

All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process. Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.

System calls used for message queues:

- ftok(): is use to generate a unique key.
- msgget(): either returns the message queue identifier for a newly created message
- queue or returns the identifiers for a queue which exists with the same key value.
- msgsnd(): Data is placed on to a message queue by calling msgsnd().
- msgrcv(): messages are retrieved from a queue.
- msgctl(): It performs various operations on a queue. Generally it is use to destroy
- message queue.

Sunstar Exam Scanner

Module - 5

9. a. Write a note on SIGCHLD, explain how SIGCLD and SIGCHLD differ and how SIGCLD is handled. (10 Marks)

Ans. Two signals that continually generate confusion are SIGCLD and SIGCHLD. First, SIGCLD (without the H) is the System V name, and this signal has different semantics from the BSD signal, named SIGCHLD. The POSIX.1 signal is also named SIGCHLD.

The semantics of the BSD SIGCHLD signal are normal, in that its semantics are similar to those of all other signals. When the signal occurs, the status of a child has changed, and we need to call one of the wait functions to determine what has happened. System V, however, has traditionally handled the SIGCLD signal differently from other signals. SVR4- based systems continue this questionable tradition (i.e., compatibility constraint) if we set its disposition using either signal or sigset (the older, SVR3-compatible functions to set the disposition of a signal). This older handling of SIGCLD consists of the following.

If the process specifically sets its disposition to SIG_IGN, children of the calling process will not generate zombie processes. Note that this is different from its default action (SIG_DFL). Instead, on termination, the status of these child processes is discarded. If it subsequently calls one of the wait functions, the calling process will block until all its children have terminated, and then wait returns 1 with errno set to ECHILD. (The default disposition of this signal is to be ignored, but this default will not cause the preceding semantics to occur. Instead, we specifically have to set its disposition to SIG_IGN.)

POSIX.1 does not specify what happens when SIGCHLD is ignored, so this behavior is allowed. The Single UNIX Specification includes an XSI extension specifying that this behavior be supported for SIGCHLD.

4.4BSD always generates zombies if SIGCHLD is ignored. If we want to avoid zombies, we have to wait for our children. FreeBSD 5.2.1 works like 4.4BSD. Mac OS X 10.3, however, doesn't create zombies when SIGCHLD is ignored.

With SVR4, if either signal or sigset is called to set the disposition of SIGCHLD to be ignored, zombies are never generated. Solaris 9 and Linux 2.4.22 follow SVR4 in this behavior.

With sigaction, we can set the SA_NOCLDWAIT flag to avoid zombies. This action is supported on all four platforms: FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3, and Solaris 9.

If we set the disposition of SIGCLD to be caught, the kernel immediately checks whether any child processes are ready to be waited for and, if so, calls the SIGCLD handler.

b. What are the different conditions to kill process using pid. (06 Marks)

Ans. There are four different conditions for the pid argument to kill.

pid > 0	The signal is sent to the process whose process ID is pid.
pid == 0	The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal. Note that the term all processes excludes an implementation-defined set of system processes. For most UNIX systems, this set of system processes includes the kernel processes and init (pid 1).
pid < 0	The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal. Again, the set of all processes excludes certain system processes, as described earlier.
pid == 1	The signal is sent to all processes on the system for which the sender has permission to send the signal. As before, the set of processes excludes certain system processes.

c. Write a program to execute read command using longjmp function. (04 Marks)

Ans.

```
#include "apue.h"
#include <setjmp.h>
static void sig_alrm(int);
static jmp_buf env_alrm;
int main(void)
{
  int n;
  char line[MAXLINE];
  if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
  if (setjmp(env_alrm) != 0)
        err_quit("read timeout");
  alarm(10);
  if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
  alarm(0);
  write(STDOUT_FILENO, line, n);
  exit(0);
}
static void sig_alrm(int signo)
{
  longjmp(env_alrm, 1);
}
```

Sunstar Exam

OR

10. Write a short note on following
    i) POSIX.1b Timers
    ii) Signal Masks
    iii) sigsetjmp
    iv) siglongjmp
    (20 Marks)

Ans. **i) POSIX.1b Timers**

The POSIX-TIMERS API is a compatible implementation of the POSIX 1003.1 real-time clock/timer API. This feature is simply a library that might or might not be linked with an application. It is not a feature that can be turned on or off when configuring a system.

POSIX Timers API

The POSIX timers API is summarized in the following table:

Function	Description
clock_settime()	Set clock to a specified value
clock_gettime()	Get value of clock
clock_getres()	Get resolution of clock
nanosleep()	Delay the current thread with high resolution
timer_create()	Create a timer
timer_delete()	Delete a timer
timer_settime()	Set and arm or disarm a timer
timer_gettime()	Get remaining interval for an active timer
timer_getoverrun()	Get current overrun count for a timer

ii) Signal Masks

Sometimes we need to block some signals, so that critical sections are not interrupted.

Every process maintains a signal mask telling which signals are blocked.

If a signal type is blocked, and signals of this type are received, they are suspended until process termination or until the signal type is unblocked.

Signal masks are stored in the data type sigset_t

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
/* return 0 if OK, -1 on error */
int sigismember(const sigset_t *set, int signo);
/* returns 1 if true, 0 if false */
```

The above are used to set the set value, not to set the process signal mask.

Call sigemptyset() or sigfillset() at least once.

sigset_t is guaranteed to be able to hold all signals supported by the UNIX implementation.

Sunstar Exam Scanner


```
#include <signal.h>
int sigpending(sigset_t *set);
/* returns 0 if OK, -1 on error      */
```

sigpending() tells us what signals are blocked and currently pending.

The list of signals is returned inside set.

Use sigismember() to find out what signals are present in set.

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
/* returns 0 if OK, -1 on error */
```

SIG_BLOCK (union)

SIG_UNBLOCK (intersection)

SIG_SETMASK (equality)

If oset is non-NULL, the old signal mask is returned in it.

set defines the signals we want to block or unblock.

If there are any pending signals, and we unblock it with sigprocmask(), one of these signals is received before sigprocmask() returns.

iii) sigsetjmp

sigsetjmp – set jump point for a non-local goto

Syntax:

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int savemask);
```

A call to sigsetjmp() saves the calling environment in its env argument for later use by siglongjmp(). It is unspecified whether sigsetjmp() is a macro or a function. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name sigsetjmp the behaviour is undefined.

If the value of the savemask argument is not 0, sigsetjmp() will also save the current signal mask of the calling thread as part of the calling environment.

All accessible objects have values as of the time siglongjmp() was called, except that the values of objects of automatic storage duration which are local to the function containing the invocation of the corresponding sigsetjmp() which do not have volatile-qualified type and which are changed between the sigsetjmp() invocation and siglongjmp() call are indeterminate.

An invocation of sigsetjmp() must appear in one of the following contexts only:

the entire controlling expression of a selection or iteration statement

one operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement

the operand of a unary (!) operator with the resulting expression being the entire controlling expression of a selection or iteration

the entire expression of an expression statement (possibly cast to void).

iv) siglongjmp

siglongjmp - non-local goto with signal handling

Syntax:

Sunstar Exam Scanner

#include <setjmp.h>
void siglongjmp(sigjmp_buf env, int val);
The siglongjmp() function restores the environment saved by the most recent invocation of sigsetjmp() in the same thread, with the corresponding sigjmp_buf argument. If there is no such invocation, or if the function containing the invocation of sigsetjmp() has terminated execution in the interim, the behaviour is undefined.
All accessible objects have values as of the time sigsetjmp() was called, except that the values of objects of automatic storage duration which are local to the function containing the invocation of the corresponding sigsetjmp() which do not have volatile-qualified type and which are changed between the sigsetjmp() invocation and siglongjmp() call are indeterminate.
As it bypasses the usual function call and return mechanisms, siglongjmp() will execute correctly in contexts of interrupts, signals and any of their associated functions. However, if siglongjmp() is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behaviour is undefined.
The siglongjmp() function will restore the saved signal mask if and only if the env argument was initialised by a call to sigsetjmp() with a non-zero savemask argument.
The effect of a call to siglongjmp() where initialisation of the jmp_buf structure was not performed in the calling thread is undefined.

Sunstar Exam Scanner