

Operating Systems

LAB I

Introduction to the UNIX

Part 1: Directory Structure and Manipulation

UNIX has a hierarchical file system. That means that all directories are based upon a root directory (in UNIX it is the `/` directory). This is illustrated in Figure I.

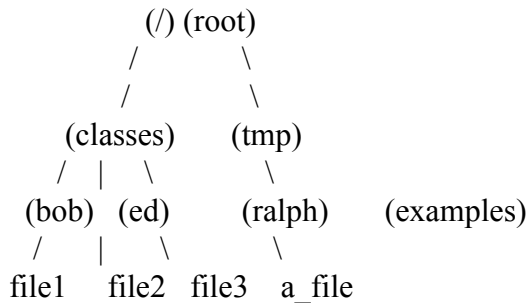


Figure I

A pathname identifies a particular file or directory uniquely. An absolute pathname starts at the root and names each directory along the way to the destination directory or file. An example is `(/tmp/examples/a_file)`. This specifies a file called `a_file` which exists in the `examples` subdirectory. A relative pathname tells how to reach the destination from the current directory. If you were in the `/tmp` directory, the path to `a_file` would be `examples/a_file`.

Every user has a home directory in UNIX. It is the directory you start off in when you first log in. In order to specify that directory when referencing it as a file a user can use the `~` key. `~/files/file` is a file located in the `files` subdirectory located in the user's home directory. In order to reference another user's home directory use `~username` (an example is `~bob/file1`).

Other special reference variables are `'.'` and `'..'`. The `'.'` reference refers to the current directory so if you are in the `/tmp` directory and you want to reference the `example` subdirectory you could use `./examples`. The `'..'` directory refers to the directory to which the current directory is a subdirectory of. If you are in the `/classes/bob` directory then `..` refers to the `/classes` directory.

Commands to use on directories and files:

`cd arg`

The change directory command. When taken with no arguments it means to change directory to the users home directory. To change to the `examples` directory you would use `cd /tmp/examples`. This accepts all wildcards and special commands like `'.'` and `'..'`

mkdir *name*

Creates a directory in the spot specified by *name*. If you wish to make a directory called *my_dir* in your current directory then the command would be **mkdir *my_dir***.

rmdir *name*

Removes a directory in the spot specified by name.

NOTE:

Directory and file permissions are very important to UNIX security because of the multiple users on the file system. Each file and directory has an owner, belongs to a user group, and contains permission bits. In order to see the permission bits of all files in a directory a user can type **ls -l** or to get a specific file a user can type **ls -l *file***. An example is

```
% ls -l /classes/bob/file1
```

Permission bits are displayed in the form: **-rwxr-xr-- 1 bob projx 538 May 24 10:12 /classes/bob/file1**

The first field (in this case -) indicates that it is a file. If it were a directory, it would be indicated with a **d**. The next three fields are the permission bits for the owner, the next three are for the group and the third three are for the rest of the world. These three groups specify whether the user has read, write, or execute/search privileges. In this case the user has all privileges. The group **projx** can read and execute the file and the world can just read the file. The owner of a directory or file can change the permissions on a file by using the **chmod** command. All users should read how to change permissions by typing in **% man chmod**

This gives you the UNIX manual page which explains the **chmod** command. You can use the **man** command to get info on most unix and c commands. The syntax is **man *command***.

Questions:

(All questions refer to **Figure I** as the file structure)

- 1) You are the user bob and are in your home directory, the */classes/bob* directory. Give 5 ways to reference the file *file3*. You must use Absolute, Relative and the '.' and '..' reference methods at least once.
- 2) If you are logged into **DCSE** and you want to go to your TA's home directory how would you do it without knowing anything about the DSCE departments file structure? (Your TA's username is *torfanos*)
- 3) If you are user bob, how would you change the permission on *file1* to just execute for user and group and no privileges for the world?

Part II: File Redirection and Pipes

Standard output is the data that is usually displayed on the screen. We may send standard output to a file by using the **command > *file*** syntax. We say that > redirects the standard output. **command** is a single command (or command group) and *file* is any valid file pathname. If you want to list what you have in your directory and save it to a file called *dir_list* you would enter **ls > *dir_list***. You can append data to a file using the >> command. To add the list of items in the root directory to your directory list you could use **ls / >> *my_dir***.

Input redirection is using a file as the input for a command. The syntax is **command < file**. An example of this is if Bob wants to mail Ed a copy of *file1*; he would enter **mail ed < file1**.

Sequences of events often require a temporary file to hold intermediate results. For example if Bob entered the commands:

```
% ls > dir_contents
```

```
% mail ed < dir_contents
```

```
% rm dir_contents (deletes the file)
```

Then Ed would get a copy of Bob's directory. We can eliminate the need for a temporary file by using a pipe to connect the output of **ls** to the input of **mail**.

The pipe symbol is | in UNIX. The solution would be **ls | mail ed**. The general format of a pipe is:

```
command1 | command2 | command3 | ... | commandn
```

Questions:

(All questions refer to **Figure I**)

1) The command **wc** counts words, characters or lines. The syntax is:

```
wc [options] [file(s)]
```

options:

```
-l counts # of lines in file
```

```
-c counts # of bytes in file
```

```
-w counts # of words in file
```

If *file2* is a list of users with more than one on each line, how would Ed (from his home directory) store the number of users in a file called *num_users*?

2) The command **sort -d file** displays a file in dictionary order. The command **more (more file)** displays the contents of a file one page at a time.

Show how Bob would display the sorted contents of his home directory one page at a time.

Part 3: Wildcards and Special Characters

When describing a file a user may use wildcards to help name more than one file. This is called filename expansion and the *cs*h (and *tc*sh which is running on the ee systems and *cs* systems) supports these characters:

?	Matches any single character
[list]	Matches any character in list.
[lower-upper]	Matches any character in the range between lower and upper (inclusive)
*	Matches any pattern (including null)

Examples:

% ls *.c	lists all files which end in .c
% ls file*	list all files that start with file (includes file by itself)
% ls program.?	list all files that start with program. and have a one letter suffix.

% ls file[1-2] list file1 and file2.

Special Characters that the shell interprets first:

**** Dereferences the following character (used to use things with * or ? in the name).

\$ Variable Identifier

; Ends a command

Questions:

These questions are based on a directory which has the following files in it:

file1 file2 file3 file4 afile file.c file.s file.so file.o farm.c farm.co

NOTE: All expansion solutions should be of the format **command file(s)**. There should only be one argument following command and in order to get multiple files you must use wildcards.

- 1) How would you list files these files (*file2, file3, file4*)? (HINT use **ls**)
- 2) How would you remove, using the **rm file(s)** command, *file.c* and *file.s* (note: keep *file.o*)?
- 3) How would you remove any file which contains the word *file* in it?
- 4) How would you list all files that begin with *f* and end with a 2 letter suffix.
- 5) If you had a file which was called *hard?* , with *?* actually being a question mark. How would you reference that file ?

Operating Systems

LAB 2




















UNIX File System

Objectives:

The aim of this laboratory is to show you some of the aspects of the Unix file system. Because we will be running user-level programs, we don't have the opportunity to see the actual physical layout of blocks in each file system.

However, we can still get an idea of what the Unix file system provides to its users.

The following are the primary objectives of this lab session:

-  **Understanding UNIX files**
-  **Different types of files**
-  **Structure of the file system**
-  **File Names**
 -  Defining files with wildcard characters and regular expressions
 -  Absolute and Relative Names
-  **Access Permissions**
 -  Understanding, Displaying and Changing access permissions
 -  Default access permissions
-  **Working with Files and Directories**
 -  Creating, removing and Displaying file
 -  Determining file type
 -  List the files in a directory
 -  Copying files
 -  Making, copying and Removing directories
 -  Changing to another directory
 -  Finding a file
 -  Searching and Sorting the contents of a file
 -  Linking + Moving files and directories

Understanding UNIX files

A file system is a logical method for organizing and storing large amounts of information in a way, which makes it easy to manage. The file is the smallest unit in which information is stored. The UNIX file system has several important features.

Files do not actually reside inside directories. **A directory is a file that contains references to other files.** The directory holds two pieces of information about each file:

- ☛ its **filename**
- ☛ an **inode** number which acts as a pointer to where the system can find the information it needs about this file.

Filenames are only used by the system to locate a file and its corresponding inode number. This correspondence is called a link.

To the system, the file is the inode number. Multiple filenames can be used to refer to the same file by creating a link between an inode and each of the filenames.

File Metadata

Every operating system keeps information about files: their name, their size, etc. This is known as **file metadata**. The metadata that Unix keeps on each file is given below (with the Unix name for each piece of data):

st_dev The **device number** of the device containing the **i-node**. This tells you on what device the file is stored.

st_ino The **i-node number**. Each file has a unique **i-node number** (that is, unique on that particular device).

st_mode The **16-bit protection** for the file.

st_nlink The number of names **links** to this file.

st_uid The **user-ID of the file's owner**.

st_gid The **group-ID**; this and the protection affects how certain people can use the file.

st_size The current **size** of the file.

st_atime The **access time** as the number of seconds since 1970. Updated whenever the file is read, but not when a directory that appears in a path is searched.

st_mtime The **modification time**, updated when the file is written. Updated when a link is added to or removed from a directory.

st_ctime The **status-change time**, updated when the file is written or when the mode, owner, group, link count, or modification time is changed.

Different types of files

To you, the user, it appears as though there is only one type of file in UNIX - the file which is used to hold your information. In fact, the UNIX file system contains several types of file.

Directories

A directory is a file that holds other files and other directories. You can create directories in your home directory to hold files and other sub-directories.

Having your own directory structure gives you a definable place to work from and allows you to structure your information in a way that makes best sense to you.

Directories which you create belong to you - you are said to "own" them - and you can set access permissions to control which other users can have access to the information they contain.

Ordinary files

This type of file is 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.

Files which you create belong to you - you are said to "own" them - and you can set access permissions to control which other users can have access to them. Any file is always contained within a directory.

Special files

This type of file is used to represent a real physical device such as a printer, tape drive or terminal.

It may seem unusual to think of a physical device as a file, but it allows you to send the output of a command to a device in the same way that you send it to a file. For example:

```
cat scream.au > /dev/audio
```

This sends the contents of the sound file **scream.au** to the file **/dev/audio** which represents the audio device attached to the system.

The directory **/dev** contains the special files which are used to represent devices on a UNIX system.

Pipes

UNIX allows you to link commands together using a pipe. **The pipe acts as a temporary file, which only exists to hold data from one command until it is read by another.**

Unix has the following file types:

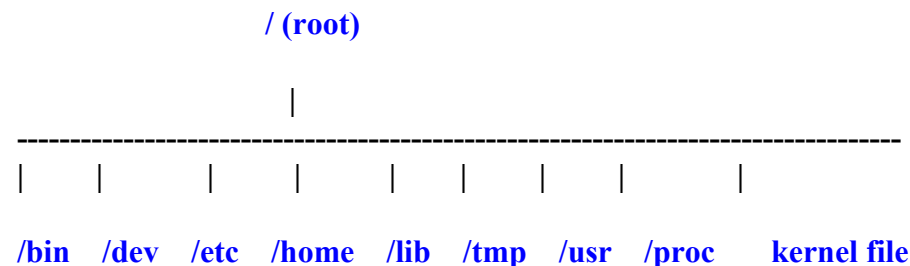
REG (-)	regular file.
DIR (d)	directory.
CHR (c)	character device (used for direct device access).
BLK (b)	block device (used for direct device access).
LNK (l)	symbolic link (more later).
FIFO (f)	named pipe.
SOCK (s)	socket.

Structure of the file system

The UNIX file system is organized as a hierarchy of directories starting from a single directory called root which is represented by a / (slash). Imagine it as being similar to the root system of a plant or as an inverted tree structure.

Immediately below the **root** directory are several **system directories** that contain information required by the operating system. The file holding the UNIX **kernel** is also here.

UNIX system directories The standard system directories are shown below. Each one contains specific types of file. The details may vary between different UNIX systems, but these directories should be common to all. Select one for more information on it.



The /bin directory contains the commands and utilities that you use day to day. These are **executable binary files** - hence the directory name **bin**.

The **/dev** directory contains special files used to represent real physical devices such as printers and terminals.

The **/etc** directory contains various commands and files which are used for system administration. One of these files - motd - contains a 'message of the day' which is displayed whenever you login to the system.

The **/home** directory contains a home directory for each user of the system.

The **/lib** directory contains libraries that are used by various programs and languages.

The **/tmp** directory acts as a "scratch" area in which any user can store files on a temporary basis.

The **/usr** directory contains system files and directories that you share with other users. Application programs, on-line manual pages, and language dictionaries typically reside here.

The **kernel** As its name implies, the kernel is at the core of each UNIX system and is loaded in whenever the system is started up - referred to as a **boot** of the system. It manages the entire resources of the system, presenting them to you and every other user as a coherent system. Amongst the functions performed by the kernel are:

- ☛ managing the machine's memory and allocating it to each process.
- ☛ scheduling the work done by the CPU so that the work of each user is carried out as efficiently as is possible.
- ☛ organizing the transfer of data from one part of the machine to another.
- ☛ accepting instructions from the shell and carrying them out.
- ☛ enforcing the access permissions that are in force on the file system.

File Names

Unix files have one or more names. **Names can consist of the characters A-Z, a-z, 0-9 and most punctuation. Spaces are not allowed; neither is the '/' character** (why not)?

Defining files with wildcard characters

Wildcard characters can be used to represent many other characters. Use them whenever you need to define a string of characters, such as a filename, for use with a command.

Useful wildcards are:

- * matches **any characters**
- ? match **any single character**
- [...] matches **any character in the enclosed list or range.**

Suppose you want to list all files that start with 'a' and ends with '.c' type the command

```
ls a*.c
```

Some Examples of using meta characters for handling files:

echo a* : Prints the names of the files beginning with a.
cat *.c : Prints all files ending with .c
rm *.* : Removes all files containing a period.
ls x* : Lists the names of all files beginning with x.
rm * : Removes all files in the current directory (Note : Be careful when you use this).
echo a*b : Prints the names of all files beginning with a and ending with b
cp ../programs/*. : Copy all files from ../programs into the current directory.
cat ? : prints all files with one character name.
echo ?? : It prints file names with two character names.
echo * : It displays all file names present in your pwd.
echo [ab]* : It displays all file names with a or b or ab both.
echo *[0-9] : Displays all file names having any digit 0-9.

Regular expressions

A regular expression is a concise way of expressing any pattern of characters.

You construct a regular expression by combining ordinary characters with one or more metacharacters: characters that have special meaning for the shell.

Matching file names with regular expressions

You can use the following metacharacters within any shell to create regular expressions that match file names.

? matches any single character
***** matches any number of any characters
[nnn] matches any of the enclosed characters
[!nnn] matches any character that is not enclosed
[n-n] matches any character in this range

Absolute and Relative Names

Files are organized hierarchically into *directories*, mainly for the benefit of the users. There are two ways of expressing the name of each file:

The **absolute** name of a file starts at the **root** of the directory tree, and gives the name of all intermediary directories leading up to the file, for example:

```
❏ /usr/local/bin/tcsh
❏ /bin/sh
❏ /home/staff/wkt
❏ /var/spool/mqueue/xxx.012643
```

Incidentally, you can't tell if any of the above names are the name of a file or a directory: **there is essentially no difference between a file and a directory in Unix.**

The **relative** name of a file starts at the **current working directory or parent working directory**.

You can use the command **pwd** to see the current working directory. If you need to go toward the root of the directory with a relative name, you can use the expression **..** to move up one level. The **..** shorthand can be used within filenames. Some examples of relative filenames could be:

```
❏ Documents/myfile.txt
❏ ../Mail/ahmed
❏ ../../staff/wkt/hello.txt
```

Home Directory

Any UNIX system can have many users on it at any one time. As a user you are given a home directory in which you are placed whenever you log on to the system.

User's home directories are usually grouped together under a system directory such as **/home**. A large UNIX system may have several hundred users, with their home directories grouped in subdirectories according to some schema such as their organizational department.

Current Directory

When you log on to the system you are always placed in your home directory. At first this is your current directory. If you then change to another directory this becomes your current directory. The command **pwd** displays the full pathname to your current directory.

Access Permissions

Every file and directory in your account can be protected from or made accessible to other users by changing its access permissions. You can only change the permissions for files and directories that you own.

Understanding Access Permissions

There are three types of permissions:

- r** **read** the file or directory
- w** **write** to the file or directory
- x** **execute** the file or search the directory

Each of these permissions can be set for any one of three types of user:

- u** **the user who owns the file (usually you)**
- g** members of the **group** to which the owner belongs
- o** all **other users**

The access permissions for all three types of user can be given as a string of nine characters:

user	group	others
r w x	r w x	r w x

These permissions have different meanings for files and directories.

Examples of access permissions

```
ls -l file1
-rw----- 2 ahmed 3287 Apr 8 12:10 file1
```

The **owner** of the file has **read** and **write** permissions and no permissions to others.

```
ls -l file2
-rw-r--r-- 2 ahmed 3287 Apr 8 12:11 file2
```

The **owner** has **read** and **write** permissions. Everyone else - the **group** and all **other users** - can **read** the file.

Displaying Access Permissions

To display the access permissions of a file or directory use the **ls** command:

```
ls -l filename or directory
```

This displays a one line summary for each file or directory. For example:

```
-rwxr-xr-x 1 ahmed staff 3649 Feb 22 15:51 prog.c
```

This first item **-rwxr-xr-x** represents the access permissions on this file. The following items represent the **number of links (1)** to it; the **username** (*ahmed*) of the person owning it; the name of the **group** (*staff*) which owns it; its **size in bytes** (*3649*); the **time** (*15:51*) and **date** (*Feb 22*) it was last changed, and finally, its **name** (*prog.c*).

Default Access Permissions

When you create a file or directory its access permissions are set to a default value. These are usually: **rw-----**

gives you **read** and **write** permission for your **files**; **no access permissions for the group or others.** **rwX-----**

gives you **read**, **write** and **execute** permission for your **directories**; **no access permissions for the group or others.**

Access permissions for your **home** directory are usually set to **rwX--X--X** or **rwXr-Xr-X**.

Changing Access Permissions

To change the access permissions for a file or directory use the command

chmod mode filename

chmod mode directory_name

The "**mode**" consists of three parts: **who** the permissions apply to, **how** the permissions are set and **which** permissions to set.

To give yourself permission to execute a file that you own:

chmod u+x file1

This gives you **execute** permission for the file "**file1**".

To give members of your **group** permission to **read** a file:

chmod g+r file2

This gives the **group** permission to **read** the file "**file2**".

To give **read** permission to **all** for a particular type of file:

chmod a+r file3

This gives **all** permission to read file "**file3**".

Setting Access Permissions Numerically There is a shorthand way of setting permissions by using **octal numbers**. **Read** permission is given the value **4**, **write** permission the value **2** and **execute** permission **1**.

r w x

4 2 1

These values are added together for any one user category:

1 = execute only

2 = write only

3 = write and execute (1+2)

4 = read only

5 = read and execute (4+1)

6 = read and write (4+2)

7 = read and write and execute (4+2+1)

So access permissions can be expressed as three digits. For example:

			user	group	others
chmod	640	file1	rw-	r--	---
chmod	754	file2	rwX	r-X	r--
chmod	664	file3	rw-	rw-	r--

Working with Files and Directories

Creating files

Create a file with the **cat** command *Type the command*

cat > name_of_file

Now type in your text. Press the <Return> key to start a new line. When you have finished typing in your text, enter Ctrl-d (Press and hold down the Ctrl key and type a "d"). This stops the cat command and returns you to the system prompt.

Text editors While using UNIX you will often want to create a text file and then change its content in some way. A text editor is a program that has been designed especially for this purpose. The easiest of all editors is the **pico** editor. Type the command

pico name_of_file

The editor will open where you can write your text or program and at bottom of editor window you will see the commands to save, quit or do other changes to text. Just follow those commands.

Removing files

To remove a file use the command: **rm filename(s)**

You cannot remove a file in another user's account unless they have set access permissions for the file which allow you to. Use the **-i** (interactive) option which makes the command prompt you for confirmation that you want to remove each file. To remove a single file: **rm help.txt** This removes the file **help.txt** from the current directory. To remove several files: **rm file1 file2 file3** This removes files file1, file2, file3 from current directory. To remove files interactively: **rm -i file** This will prompt you to confirm that you want to remove **file** from the current directory.

Answering **y** will delete the file. The file is not deleted if any other response is given.

Determining file type

The file command examines the content of a file and reports what type of file it is. To use the command enter: **file filename**

Use this command to check the identity of a file, or to find out if executable files contain shell scripts, or are binaries. Shell scripts are text files and can be displayed and edited.

Displaying files

The **cat** command is useful for displaying short files of a few lines. To display longer files use **page** or **more** that displays files page by page of 25 or so lines.

To display the contents of a file use the commands:

cat filename

page filename

more filename

To display the first **n** number of lines of a text file use the command: **head -n**

filename To display the last **n** number of lines of a text file use the command: **tail -n**

filename Note: Both the **head** and **tail** commands displays only first and last **10** lines respectively if the option of **n** is not specified.

List the files in a directory

You can use the **ls** command to list the files in a directory:

ls [option] directory_name

By combining different command options you can display as little or as much information about each file as you need.

Listing hidden files The command **ls -a**

lists all the "hidden" files that begin with a '.' (dot). All other files and directories are also listed. Every directory has two dot files, '.' and '..' which can be used in a shorthand way to refer to the current directory '.' (dot) and the parent directory of the current directory '..' (dot dot).

Using a long listing To get more information about each file and directory, use the command: **ls -l**

This gives you a long listing about each file and directory, giving information about its: **access permissions, number of links, owner, group ownership, size, date and time last modified**

Copying files

Copying files in the same directory To create an exact copy of a file use the **cp** (copy) command.

cp old_file new_file

The **old_file** is the name of the file to be copied; the **new_file** is the name of the file in which the copy is to be placed.

Copying more than one file You can use special "wildcard" characters whenever you want to copy several files that have similar filenames. Instead of entering the **cp** command followed by several filenames you can use a single filename that contains one or more wildcards.

cp file1 file2 file3 sub_directory **or**
cp file* sub_directory

Copies three files to a sub directory of the current directory.

Copying files to another directory To copy a file to another directory from your current directory give name of the source file followed by the pathname to the destination file.

cp source path_to_destination

For the destination file to have the same name as the source file use:

cp source path_to_destination_directory

To copy a file from your current working directory to a subdirectory:

cp fig2 part2/figure2

This copies the file **fig2** from your current working directory to the file **figure2** in the subdirectory **part2**.

To copy a file to the parent directory:

cp mail.txt ..

This copies the file **mail.txt** to the directory immediately above the current working directory with the same name **mail.txt**. The **..** (dot dot) is shorthand for the **parent** directory.

Copying files from another directory To copy a file from another directory to your current directory give the pathname to the source file followed by the name of the destination file.

cp path_to_source_file destination

For the destination file to have the same name as the source file use:

cp path_to_source_file .

The **.** (dot) is shorthand for the current working directory.

To copy a file from a subdirectory to the current working directory:

cp notes/note3 sect3.txt

This copies the file **note3** from the subdirectory **notes** to the file **sect3.txt** in the current working directory. A relative pathname **notes/note3** is used to define the source file.

To copy a file from another directory to the current working directory, preserving the file name: **cp /usr/lib/more.help .**

This creates a copy of the file **more.help** in the current working directory. A full pathname **/usr/lib/more.help** is used to define the source file.

Making a directory

To make a directory use the command:

mkdir directory_name

The access permissions for a directory that you create are set to a predetermined value which ensures that other users cannot get access to your directories and their contents.

To make a directory in the current directory:

mkdir specification

This creates a new directory **specification** in your current working directory.

To make a new directory in the parent directory:

mkdir ../presentations

This creates the directory **presentations** in the parent directory of the current working directory.

Removing directories

To remove a directory use the command:

rmdir directory_name

The directory must be empty before you can delete it. You will need to remove any files and subdirectories that it contains.

To remove a directory that contains files use the command:

rm -r directory_name

This deletes all the contents of the directory including any subdirectories.

Changing to another directory

To change your current working directory use the command:

```
cd pathname
```

where `pathname` specifies the directory that you want to move to. The `pathname` can be given as either a [full pathname](#) or a [relative pathname](#).

To move down one level to a subdirectory: `cd Firstyear`

This moves you down one level from your current directory to the subdirectory `Firstyear`.

To move up one level of the directory tree: `cd ..`

Every directory contains a hidden directory `..` (dot dot) that is a shorthand name for this directory's parent directory. Using this shorthand name enables you to move up the directory tree very quickly without having to enter long pathnames.

To move to another directory using a relative pathname: `cd ../Secondyear`

This moves you up one level in the directory tree and then moves you into the subdirectory `Secondyear`.

Copying directories

To copy a directory use the command:

```
cp -r directory1 directory2
```

This copies `directory1` and everything that it contains to `directory2`. The directory is created if it does not exist. If `directory2` does exist then `directory1` is created as a subdirectory within it.

Displaying the pathname to the current directory

To display the pathname to your current directory use the command:

```
pwd
```

This command has no options.

Moving files and directories

To move files and directories from one place to another use the `mv` (move) command:

```
mv filename1 filename2
   directory1 directory2
   filename  directory
```

Note: You can also change the name of a file or directory by moving it.

To rename a file:

```
mv junk precious
```

This renames the file junk as the file precious.

Finding a file

To locate a file in the file system , use the `find` command.

```
find pathname -name filename -print
```

The `pathname` defines the directory to start from. Each subdirectory of this directory will be searched. The `-print` option must be used to display results. You can define the filename using wildcards. If these are used, the filename must be placed in 'quotes'.

To find a single file below the current directory.

```
find . -name program.c -print
```

This displays the pathname to the file `program.c` starting from the current directory. If the file is not found nothing is displayed.

To find several files below the current directory:

```
find . -name '*.c' -print
```

This displays the pathname to any file with the extension `.c` which exists below the current directory.

Searching the contents of a file

To search a text file for a string of characters or a regular expression use the command:

```
grep pattern filename(s)
```

Using this command you can check to see if a text file holds specific information. `grep` is often used to search the output from a command. Any regular expression containing one or more `special characters` must be `quoted` to remove their meaning.

To search a file for a simple text string:

```
grep copying help
```

This searches the file `help` for the string `copying` and displays each line on your terminal.

To search a file using regular expression:

```
grep -n '[dD]on\t' tasks
```

This uses a regular expression to find and display each line in the file `tasks` that contains the pattern `don't` or `Don't`. The line number for each line is also displayed.

The expression is quoted to prevent the shell expanding the metacharacters `[`, `]` and `'`. Double quotes are used to quote the single quote in `dDon't`.

Sorting the content of a file

The `sort` command is a simple database tool. It allows you to specify the field to be sorted on and the type of sort to be carried out on it.

`sort filename`

Lines are sorted into the following order: lines starting with numbers come first, followed by lines starting with upper-case letters, which are followed by lines starting with lower-case letters and finally symbols such as % and !.

Sorting on a specific field

Lines are sorted character by character, starting at the first character in the line. You can also sort the contents of a file on a specific part of each line.

Each line of text is a series of fields - words and other characters - separated from each other by a delimiter character - the spaces between the words.

Defining the sort field

The first field of each line starts at 0 (zero); the second is 1 (one) and so on. To define which field to sort on you give the position of the field at which to start the sort followed by the position at which to end the sort.

The position at which to start the sort is given as the number of fields to skip to get to this position. For example +2 tells sort to skip the first two fields.

The position at which to stop the sort is given as the number of the field at the end of which the sort stops. For example -3 tells sort to stop the sort at the end of field three.

To sort on the third field of a line use the definition: +2 -3

To sort on the fields 5 and 6: +4 -6

To sort a file on field 2 (the third word):

`sort +2 -3 names`

This sorts the file `names` on the third word of each line.

Linking files and directories

To link files and directories use the command:

`ln source linkname`

Making a link to a file or directory does not create another copy of it; it simply makes a connection between the source and the linkname.

Using symbolic links Your files (and directories) may be located on several different file systems. To link files that are in different file systems you need to make a symbolic link.

To make a symbolic link use the command:

`ln -s source linkname`

To make several links to a file in different directories:

`ln part1.txt ../helpdata/sect1 /public/helpdoc/part1`

This links `part1.txt` to `../helpdata/sect1` and `/public/helpdoc/part1`.

Linking directories

To link one or more directories to another directory use the command:

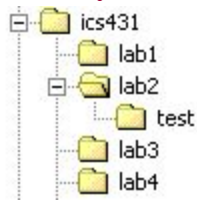
```
ln -s directory_name(s) directory_name
```

The use of the -s option indicates that this is to be a symbolic link. **Only the super-user is allowed make hard links between directories.** As a user you are restricted to creating symbolic links between directories using the -s option.

Assignment Problems:

1. Run all the commands given in the Lab Notes, and observe the output for each command.

2. Create a directory named **ics431** under your home directory. Then create directories named **lab1 lab2 lab3 ...** inside the **ics431** directory. Also create a directory named **test** inside the directory **lab2**. **Write all the commands.**



3. a. Now create a C program file named **myprogram.c** which displays "C is a programming Language". This file should be in **lab2** directory.

b. Record (using script command) the following operations in a file called **rec** - in the directory **test**.

- i. Compile the program
 - ii. Execute the program and see the output
 - iii. stop recording and see the **rec** file contents using **cat** command
 - iv. redirect the output of this program to a file called **out**
 - v. **What is the content of out and rec. Are They same?**
4. Go to the **ics431** directory and list all the directory names starting with **l**. **Write the commands.**
5. From **ics431** directory, create a soft link to the **test** directory in the name **linktest**.
- a. Go to **linktest** directory and display the files. **What Files are displayed?**
 - b. From there go to the parent directory. **Which parent are you getting? State reasons.**
6. Try to open the created file **myprogram.c** in the **notepad** of your desktop computer. This can be done by using **ftp**. Modify the program in notepad to print "Now I like know Unix and windows OS". and execute in the Unix environment (again **ftp** is needed). **Write all the commands to do this.**
7. Move the file **rec** to the directory **lab1** and delete the directory **test** and observe what had happened to **linktest**. **What is it pointing to?**
8. Record the following:
- a. Go to directory **lab1**
 - b. change the modes of all files to [read exec to owner & group and only execute to others]

- c. Try to delete **rec** file and observe the o/p. **Write the o/p**
9. Display a file containing all the full names (in sorted order) of the users currently logged in to the Unix server
Write the commands to do this.
10. Make a copy of the directory **ics431** in the same level and name it as **CopyOfics431**. (All subdirectories and files inside should be copied.)

Operating Systems LAB 3

Introduction to C Programming

Objective:

To gain experience with

1. Writing simple c programs with more than one function (Parameters passed by value)
2. Basic concepts of Pointers in C
3. Passing parameters to the function by pointers.
4. Using Arrays in C
5. Using Structures in C
6. Use of Linked List

Purpose:

To gain experience about C programming

1. A Simple C program with more than one function (Parameters passed by value)

The following program reads two numbers and finds the sum of those two numbers

Program

```
#include <stdio.h>

int add(int x, int y){
    return x+y;
}

main (void) {
    int    a, b, x;
    printf ("Enter the values for integers a and b\n") ;
    scanf ("%d %d", &a, &b);
    printf ("The result is = %d\n", add(a,b)) ;
}
```

sample output

```
colonel > ./passval
Enter the values for integers a and b
10 120
The result is = 130
```

2. Basic concepts of Pointers in C

Every variable in C has a name (variable name), a memory location (to store the data), and an address.

For the variable declaration: `int num;` **num**----->variable name

Memory

`8152` -----> address

A variable used to store the address value is called as the Pointer. It can be defined as `int *ptr;` The following program demonstrates about the pointer variable, * and & operators.

```
#include <stdio.h>
int main (void) {
    int a;    int *p;
    printf("Enter an Integer: ");
    scanf("%d",&a);
    p=&a;
    printf("The value of the variable a is %d\n",a);
    printf("The address of the variable a is %x\n",&a);
    printf("The value of variable p is %x\n",p);
    printf("The value pointed by p is *P = %d\n",*p);
    printf("The address of p is %x\n",&p);
    return(0);
}
```

3. Passing parameters to function by pointers

The following program reads two values and adds the two numbers, the result is passed by pointer

```
Program
#include <stdio.h>
void add(int x, int y, int *z){
    *z = x+y; }
main (void) {
    int a, b, r;
    printf ("Enter the values for integers a and b\n") ;
    scanf ("%d %d", &a, &b);
    add(a,b,&r);
    printf ("The result is = %d\n", r);
}
```

Sample output

```
Enter the values for integers a and b
100
200
```


The result is = 300

4. Using Arrays in C

Example:

```
/******  
Create three arrays. Read data into the first two of them. Subtract each  
element in the first array from the corresponding element in the second  
array. Store the differences in the third array. Print all the arrays.  
*****/  
#include<stdio.h>  
#define MAX_SIZE 5  
void main() {  
    int first[MAX_SIZE], second[MAX_SIZE], diff[MAX_SIZE], i;  
    printf("\nEnter %d data items for first array : ", MAX_SIZE);  
    for(i=0; i<MAX_SIZE; i++) // input first array  
    { // input first array  
        scanf("%d", &first[i]);  
    }  
    printf("\nEnter %d data items for second array : ", MAX_SIZE);  
    for(i=0; i<MAX_SIZE; i++) // input second array  
        scanf("%d", &second[i]);  
    for(i=0; i<MAX_SIZE; i++) // compute the differences  
        diff[i]=second[i] - first[i];  
    printf("\n\nOutput of the arrays : ");  
    for(i=0; i<MAX_SIZE; i++) // output the arrays  
        printf("\n\n%5d %5d %5d", first[i], second[i], diff[i]);  
} // end of main
```

5. Using Structures in C

Structures

Structure is a collection / group of different / same variables.

Example:

```
Program  
#include<stdio.h>  
main(){  
    struct student {  
        char name[20];  
        int id;  
    };  
    struct student s1, s2, s3;  
    printf("Please enter the student name, and id\n");  
    scanf("%s %d", &s1.name, &s1.id);  
    scanf("%s %d", &s2.name, &s2.id);
```

```
scanf("%s %d", &s3.name, &s3.id);
printf("\nThe student details");
printf("\n%s \t\t%d",s1.name,s1.id);
printf("\n%s \t\t%d",s2.name,s2.id);
printf("\n%s \t\t%d",s3.name,s3.id);

}
```

Sample output

```
colonel > ./structure
```

Please enter the student name, and id

ahamed 9876

Ali 9979

Yahya 9988

The student details

ahamed 9876

Ali 9979

Yahya 9988

6. Use of Linked List in C

Using malloc to obtain memory at run-time.

☛ Memory can be allocated dynamically (at run-time) using the function **malloc()** – accessible through **<stdlib.h>**

☛ The allocation is made from a special memory area called the **heap**.

☛ The function, malloc() returns a pointer (address) to the allocated storage.

☛ However, malloc() does not associate any type to the pointer it returns – it is said to be **void**.

☛ For the pointer to be useful, it must be associated with a type using casting.

e.g.

```
int *int_ptr;
int_ptr=(int *) malloc(2);
*int_ptr =17;
```

The above statements reserve two bytes and return the address of first byte, cast it to **int** and assign it to integer pointer **int_ptr**.

☛ Since the bytes allocated to **int** is system-dependent, it is safer to use the function **sizeof ()** to get the actual number of bytes associated with the particular type being considered.

☛ **sizeof()** is system-independent and can be used even with user-defined types.

Thus, the above statements are better represented as follows:

```
int *int_ptr;
int_ptr=(int *) malloc(sizeof(int));
```

```
*int_ptr=17;
```

☛ Note that there is no name associated with the memory obtained by malloc. It can only be accessed as ***int_ptr**. It is sometimes called **anonymous** variable.

☛ Thus, should **int_ptr** be given another address, the location (returned by malloc) will be lost. It can neither be accessed by the program nor by the system. It is said to be a **lost** object.

When we no longer need a dynamic variable, we can return the storage it occupies using the free() function.

e.g. **free(int_ptr);**

7. Exercises

1. Run all the above example problems and try to understand the concepts.
2. Write a complete menu driven program to do the following:
 - Build a linked list to save a list of names. Name will not exceed 50 characters.
 - Write a function add to append a new name to the list. The function prototype is given as
void add (list *head, char *newname);
 - Write a function search to look for a given name in the list. If that name is found in list then the function should return true, otherwise, return false.
 - Write a main method to test your two functions.

In C language, the boolean type and the boolean literals (true, false) are not defined. We can define these in our program as follow:

```
typedef enum {false = 0, true} boolean;
```

The skeleton of your program should look like the following:

Program

```
#include <stdio.h>
#include <stdlib.h>
typedef struct list {
    ...
    ...
} list;
typedef enum {false=0, true} boolean;
void add (list *, char *);
boolean search (list *, char *);
int main()
{
    ...
    ...
}
void add (list *head, char *newname)
{
    ...
}
boolean search (list *head, char *name)
```

$\{ \dots \}$

Operating Systems

LAB 4

SHELL Programming (Part I)

Objectives:

The aim of this laboratory is to learn and practice SHELL scripts by writing small SHELL programs.

The following are the primary objectives of this lab session:

- 🌐 **Understanding what is a SHELL script**
 - 📖 What is a SHELL script
 - 📖 Different kinds of SHELLs in UNIX
- 🌐 **Why and where it is used**
- 🌐 **First simple SHELL script**
- 🌐 **SHELL variables**
 - 📖 User defined variables
 - 📖 System variables
 - 📖 Read only variables and wiping out variables
 - 📖 Assigning values to variables
 - 📖 Reading input

Understanding what is a SHELL script

Shell

A shell is a command line interpreter. It takes commands and executes them. As such, it implements a programming language. Three most widely used shells in UNIX are Bourne shell, C shell, and Korn shell.

Shell scripts

A shell script or a shell program is a series of commands put in a file and executed by the Shell. We will use shell is used to create shell scripts.

Why and where it is used

Why Shell scripts?

Since the user cannot interact with the kernel directly, Shell programming skills are a must to be able to exploit the power of UNIX to the fullest extent. A shell script can be used for variety of tasks and some of them are listed below.

Uses of Shell scripts

1. Customizing your work environment. For Example Every time you login, if you want to see the current date, a welcome message, and the list of users who have logged in you can write a shell script for the same.
2. Automating your daily tasks. For example, to back up all the programs at the end of the day.
3. Automating repetitive tasks.
4. Executing important system procedures, like shutting down the system, formatting a disk, creating a file system etc.
5. Performing some operations on many files.

First simple SHELL script

Create a file named SS1 (shell script 1) and type the following as content.

Example1:

```
# SS1
# Usage: SS1
ls
who
pwd
```

Save the file and type the file name in command line and the file is executed as follows.

```
console> SS1
SS1: Command not found.
```

The reason for this message is, the path of this command should be specified.

```
console> ./SS1
SS1: Permission Denied.
```

Now you do not have the permission to execute the file. That is the default permission given for any file is with out execute permission. How to know the default permission? Type the command **umask**. This command will give the masked permissions of a file while creation.

Change the permissions using **chmod** command and execute the file again.

SHELL variables

The variables in the Shell are classified as

- **User defined variables:** defined by the user for his use (e.g. age=32).
- **Environmental variables:** defined by shell for its own operations (PATH, HOME, TERM, LOGNAME, PS1, SHELL etc.).
- **Predefined variables:** reserved variables used by the shell and UNIX commands for specifying the exit status of command, arguments to the shell scripts, the formal parameters etc.

Examples:

```
name=Ali                #variable name is assigned a value Ali
echo $name              #Ali will be displayed
echo Hello $name ! , Welcome to $HOME #see output of this in your computer
```

To read [standard input](#) into a shell script use the **read** command. For example:

Example2:

```
# SS2
# Usage: SS2
# An interactive shell script
echo What is your name\?
read name
echo Hello $name. Assalam-o-Alaikum.
```

This prompts the user for input, assigns this to the variable name and then displays the value of this variable to [standard output](#).

If there is more than one word in the input, each word can be assigned to a different variable. Any words left over are assigned to the last named variable. For example:

Example3:

```
# SS3
# Usage: SS3
echo "Please enter your surname\n"
echo "followed by your first name: \c"
```

```
read name1 name2
echo "Welcome to CSE Dept., UET, $name2 $name1"
```

Example4:

```
# SS4
# This script takes two file names and copies the first file into the second one
echo "Please Enter source file name: \c"
read source
echo "Enter the target file name :\c"
read target
cp $source $target
echo file $source is copied into the $target
```

Command Substitution:

Format for command substitution is:

var=`command` (where ` ` is back quote)

Examples:

```
echo `date`      # It will display the output of date command
echo there are `who | wc -l` users working on the system # see output of this
```

Arithmetic in SHELL script

Various forms for performing computations on shell variables using **expr** command are:

expr val_1 op val_2 (Where **op** is operator)

expr \$val_1 op \$val_2

val_3=`expr \$val_1 op \$val_2`

Examples:

```
expr 5 + 7      # Gives 12
expr 6 - 3      # Gives 3
expr 3 \* 4     # Gives 12
expr 24 / 3     # Gives 8
sum=`expr 5 + 6`
echo $sum       # Gives 11
```

Example 5:

```
a=12
b=90
echo sum is $a + $b      # Will display sum is 12 + 90
echo sum is `expr $a + $b` # Gives sum is 102
```


Run all the programs given in the Lab Notes, and observe the output for each program.

Practice different examples and show them in lab report.








Operating Systems LAB 5

SHELL Programming (Part II)

Objectives:

The aim of this laboratory is to learn and practice SHELL scripts by writing small SHELL programs.

The following are the primary objectives of this lab session:

-  **SHELL keywords**
-  **Arithmetic in SHELL script**
-  **Control Structures**
 -  Decision control
 -  Repetition control
-  **More UNIX commands**
-  **Executing commands during login time**

Handling shell variables

The shell has several variables which are automatically set whenever you login. The values of some of these variables are stored in [names](#) which collectively are called your user environment. Any name defined in the user environment, can be accessed from within a shell script. To include the value of a shell variable into the environment you must [export](#) it.

Special shell variables/Pre-defined shell variables (Parameters to shell scripts):

There are some variables which are set internally by the shell and which are available to the user:

Name	Description
\$1 - \$9	these variables are the positional parameters (Positional parameters).
\$0	the name of the command currently being executed (The command name).
\$#	number of positional arguments given to this invocation of the shell.
\$\$	the process number of this shell
\$*	a string containing all the arguments to the shell, starting at \$1 (All parameters).
\$@@	same as above, except when quoted.

Passing arguments to the shell

Shell scripts can act like standard UNIX commands and take arguments from the command line. Arguments are passed from the command line into a shell program using the positional parameters \$1 through to \$9. Each parameter corresponds to the position of the argument on the command line. The positional parameter \$0 refers to the command name or name of the executable file containing the shell script. Only nine command line arguments can be accessed, but you can access more than nine [using the shift](#) command.

All the positional parameters can be referred to using the special parameter \$*. This is useful when passing filenames as arguments.

Examples of passing arguments to the shell

Write shell script which will accept 5 numbers as parameters and display their sum. Also display the contents of the different variables in the script.

Example1:

```
# Usage: SS1 param1 param2 param3 param 4 param5
# Script to accept 5 numbers and display their sum.
echo the parameters passed are : $1, $2, $3, $4, $5
echo the name of the script is : $0
echo the number of parameters passed are : $#
sum=`expr $1 + $2 + $3 + $4 + $5`
echo The sum is : $sum
```

Why need of shift command?

If more than 9 parameters are passed to a script, it is not possible to refer to the parameters beyond the 9th one. This is because shell accepts a single digit following the dollar sign as a positional parameter definition.

The shift command is used to shift the parameters one position to the left. On the execution of shift command the first parameter is overwritten by the second, the second by third and so on. This implies, that the contents of the first parameter are lost once the shift command is executed.

Example of shift:

Write a script which will accept different numbers and finds their sum. The number of parameters can vary.

```
sum=0
while [ $# -gt 0 ]
do
    sum=`expr $sum + $1`
    shift
```

```
done
echo sum is $sum
```

Here, the parameter \$1 is added to the variable sum always. After shift, the value of \$1 will be lost and the value of \$2 becomes the value of \$1 and so on.

The above script can also be written without using the shift command as:

```
for i in $*
do
    sum=`expr $sum + $i`
done
```

Usually only nine command line arguments can be accessed using positional parameters. The shift command gives access to command line arguments greater than nine by shifting each of the arguments.

The second argument (\$2) becomes the first (\$1), the third (\$3) becomes the second (\$2) and so on. This gives you access to the tenth command line argument by making it the ninth. The first argument is no longer available.

Successive shift commands make additional arguments available. Note that there is no "unshift" command to bring back arguments that are no longer available!

Another Example of using the shift Command

To successively shift the argument that is represented by each positional parameter:

Example 3:

```
#Usage: SS3 param1 param2 param3 param4 param5 param6 param7 param8
      param9 param10 param11 param12
echo "arg1=$1 arg2=$2 arg3=$3"
shift
echo "arg1=$1 arg2=$2 arg3=$3"
shift
echo "arg1=$1 arg2=$2 arg3=$3"
shift
echo "arg1=$1 arg2=$2 arg3=$3"
```

Control Structures

Every UNIX command returns a value on exit which the shell can interrogate. This value is held in the read-only shell variable \$?.

A value of 0 (zero) signifies success; anything other than 0 (zero) signifies failure.

The if statement

The if statement uses the exit status of the given command and conditionally executes the statements following. The general syntax is:

```
if test
then
    commands    (if condition is true)
```

```
else
    commands    (if condition is false)
fi
```

then, else and fi are shell reserved words and as such are only recognized after a new line or ; (semicolon). Make sure that you end each if construct with a fi statement.

Nested if statement :

```
if ...
then ...
else if ...
    ...
fi
fi
```

The **elif** statement can be used as shorthand for an **else if** statement. For example:

```
if ...
then ...
elif ...
    ...
fi
```

Test Command:

The UNIX system provides **test** command which investigates the exit status of the previous command and translate the result in the form of success or failure, i.e. either a 0 or 1.

The test command does not produce any output, but its exit status can be passed to the if statement to check whether the test failed or succeeded.

How to know exit status of any command?

All commands return the exit status to a pre-defined Shell Variable '?. Which can be displayed using the echo command.

e.g.

```
echo $?
```

If output of this is 0 (Zero) it means the previous command was successful and if output is 1 (One) it means previous command failed.

The test command has specific operators to operate on files, numeric values and strings which are explained below:

Operators on Numeric Variables used with test command:

- eq : equal to
- ne : not equals to
- gt : greater than
- lt : less than
- ge : greater than or equal to
- le : less than equal to

Examples:

```
a=12; b=23
test $a -eq $b
echo $?    # Gives 1 (one) as output.(Indicates exit status false)
```

Operators on String Variables used with test command:

= : equality of strings
!= : not equal
-z : zero length string (i.e. string containing zero character i.e. null string).
-n : String length is non zero.

Examples:

```
name="Ahmad"
test -z $name      # will return the exit status 1 as the string name is not null.
test -n $name      # will return 0 as the string is not null.
test -z "$address" # will return 0 as the variable has not been defined.
test $name = "Ali" # will return 1 as the value of name is not equal to "Ali"
```

Operators on files used with test command:

-f : the file exists.
-s : the file exists and the file size is non zero.
-d : directory exists.
-r : file exists and has read permission.
-w : file exists and has write permission.
-x : file exists and has execute permission.

Examples:

```
test -f "mydoc.doc" # Will check for the file mydoc.doc, if exists, returns 0 else 1.
test -r "mydoc.doc" # Will check for read permission for mydoc.doc
test -d "$HOME"     # Will check for the existence of the users home directory.
```

Logical Operators used with test command:

Combining more than one condition is done through the logical AND, OR and NOT operators.

-a : logical AND
-o : logical OR
! : logical NOT

Example:

`test -r "mydoc.doc" -a -w "mydoc.doc" # Will check both the read and write permission for the file mydoc.doc and returns either 0 or 1 depending on result.`

Example of using an if construct:

To carry out a conditional action:

Example 4:

```
a=10
b=20
if [ $a == $b ]
then
    echo "a is equal to b"
elif [ $a -gt $b ]
then
    echo "a is greater than b"
elif [ $a -lt $b ]
then
    echo "a is less than b"
else
    echo "None of the condition met"
fi
```

Example 5:

```
if who | grep -s student > /dev/null
then
    echo student is logged in
else
    echo student is not available
fi
```

This lists [who](#) is currently logged on to the system and [pipes](#) the output through [grep](#) to search for the username student.

The -s option causes grep to work silently and any error messages are directed to the file /dev/null instead of the [standard output](#).

If the command is successful i.e. the username student is found in the list of users currently logged in then the message student is logged on is displayed, otherwise the second message is displayed

Flow of control statements:

The Bourne shell provides several flow of control statements. Select an item for further information.

The case statement:

The **case statement** case is a flow control construct that provides for multi-way branching based on patterns.

Program flow is controlled on the basis of the *word* given. This *word* is compared with each *pattern* in order until a match is found, at which point the associated *command(s)* are executed.

```
case word in
  pattern1)      command(s) ;;
  pattern2)      command(s) ;;
  -----
  -----
  patternN)      command(s) ;;
  *) default    command ;;
esac
```

When all the commands are executed control is passed to the first statement after the *esac*. Each list of commands must end with a double semi-colon (;:).

A command can be associated with more than one pattern. Patterns can be separated from each other by a | symbol.

For example:

```
case word in
  pattern1|pattern2) command
  ... ;;
```

Patterns are checked for a match in the order in which they appear. A command is always carried out after the first instance of a pattern.

The * character can be used to specify a default pattern as the * character is the shell [wildcard](#) character.

Example 6:

```

# Display a menu of options and depending upon the user's choice,
#execute associated command
#Display the options to the users
clear
echo "1. Date and time"
echo
echo "2. Directory listing"
echo
echo "3. Users information"
echo
echo "4. Current Directory"
echo
echo "Enter choice (1,2,3 or 4) :\c"
read choice
case $choice in
1)    date;;
2)    ls -l;;
3)    who ;;
4)    pwd ;;
*)    echo wrong choice;;

```

The for statement:

The **for** loop notation has the general form:

```

for var in list-of-words
do
    commands
done

```

commands is a sequence of one or more commands separated by a newline or ; (semicolon).

The reserved words **do** and **done** must be preceded by a newline or ; (semicolon). Small loops can be written on a single line.

For example:

```
for var in list; do commands; done
```

Examples of using the **for** statement :

To take each argument in turn and see if that person is logged onto the system.

Example 7:

```

# see if a number of people are logged in
for i in $*

```



```

do
  if who | grep -s $i > /dev/null
  then
    echo $i is logged in
  else
    echo $i not available
  fi
done

```

For each username given as an argument an [if statement](#) is used to test if that person is logged on and an appropriate message is then displayed.

The while and until statements:

The while statement has the general form:

```

while command-list1
do
  command-list2
done

```

The commands in *command-list1* are executed; and if the exit status of the last command in that list is 0 (zero),

the commands in *command-list2* are executed.

The sequence is repeated as long as the exit status of *command-list1* is 0 (zero).

The until statement has the general form:

```

until command-list1
do
  command-list2
done

```

This is identical in function to the **while** command except that the loop is executed as long as the exit status of *command-list1* is non-zero.

The exit status of a while/until command is the exit status of the last command executed in *command-list2*. If no such command list is executed, a while/until has an exit status of 0 (zero).

The break and continue statements:

It is often necessary to handle exception conditions within loops. The statements break and continue are used for this.

The break command terminates the execution of the innermost enclosing loop, causing execution to resume after the nearest done statement.

To exit from *n* levels, use the command:

```

break n

```

This will cause execution to resume after the done n levels up.

The **continue** command causes execution to resume at the while, until or for statement which begins the loop containing the continue command.

You can also specify an argument $n|FR$ to continue which will cause execution to continue at the $n|FR$ th enclosing loop up.

Example of using the **break** and **continue** statements

Example 8:

```
while echo "Please enter command"
read response
do
  case "$response" in
    'done') break      # no more commands
              ;;
    "")    continue    # null command
              ;;
    *)    eval $response # do the command
              ;;
  esac
done
```

This prompts the user to enter a command. While they enter a command or null string the script continues to run. To stop the command the user enters **done** at the prompt.

Some more examples for writing shell scripts :

#SS9

```
# To show use of case statement .
echo What kind of tree bears acorns\ ?
read response
case $response in
[Oo][Aa][Kk]) echo $response is correct ;;
*) echo Sorry, response is wrong
esac
```

#SS10

```
# To show use of while statement
clear
echo What is the Capital of Saudi Arabia \?
```

```
read answer
while test $answer != Riyadh
do
echo No, Wrong please try again.
read answer
done
echo This is correct.
```

```
#SS11
# Example to show use of until statement
# Accept the login name from the user
clear
echo "Please Enter the user login name: \c"
read login_name
until who | grep $login_name
do
    sleep 30
done
echo The user $login_name has logged in
```

```
#SS12
#To show use of if statement
# Read three numbers and display largest
clear
echo "Enter the first number :\c"
read num1
```

```
echo "Enter the second number :\c"
read num2
```

```
echo "Enter the third number :\c"
read num3
```

```
if test $num1 -gt $num2
then
    if test $num1 -gt $num3
    then
        echo $num1 is the largest
    else
        echo $num3 is the largest
    fi
else
    if test $num2 -gt $num3
```

```
        then
            echo $num2 is largest
        else
            echo $num3 is the largest
        fi
    fi
```

Assignment Problems on UNIX SHELL programming

1. Run all the programs given in the Lab Notes, and observe the output for each program.
2. Write a shell script that takes a keyword as a command line argument and lists the filenames containing the keyword
3. Write a shell script that takes a command line argument and reports whether it is a directory, or a file or a link.
4. Write a script to find the number of sub directories in a given directory.
5. Write a menu driven program that has the following options.
 - 5.1. Search a given file is in the directory or not.
 - 5.2. Display the names of the users logged in.

Operating Systems

LAB 6

Process Creation and Execution

Objective:

This lab describes how a program can create, terminate, and control child processes. Actually, there are a few distinct operations involved: **creating a new child process**, and **coordinating the completion of the child process with the original program**.

What is a process? :

A **process** is basically a **single running program**. It may be a **system** program (e.g. login, update, csh) or **program initiated by the user** (pico, a.exe or a user written one). When UNIX runs a process it gives each process a unique number - a **process ID, pid**. The UNIX command **ps** will list all current processes running on your machine and will list the **pid**. The C function **int getpid()** will return the **pid** of process that called this function.

Processes are the primitive units for allocation of system resources. Each process has its own **address space** and (usually) one thread of control. **A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.**

Processes are organized hierarchically. Each process has a **parent** process which explicitly arranged to create it. The processes created by a given parent are called its **child** processes.

A child inherits many of its attributes from the parent process.

Every process in a UNIX system has the following attributes:

- **some code**
- **some data**
- **a stack**
- **a unique process id number (PID)**

When UNIX is first started, there's only one visible process in the system. This process is called **"init"**, and its **PID** is **1**. The only way to create a new process in UNIX is to duplicate an existing process, so **"init"** is the ancestor of all subsequent processes. When a process duplicates, the parent and child processes are identical in every way except their **PIDs**; the child's code, data, and stack are a copy of the parent's, and they even continue to execute the same code. **A child process may, however, replace its code with that of another executable file, thereby differentiating itself from its parent.** For example, when **"init"** starts executing, it quickly duplicates several times. Each of the

duplicate child processes then replaces its code from the executable file called “**getty**” which is responsible for handling user logins.

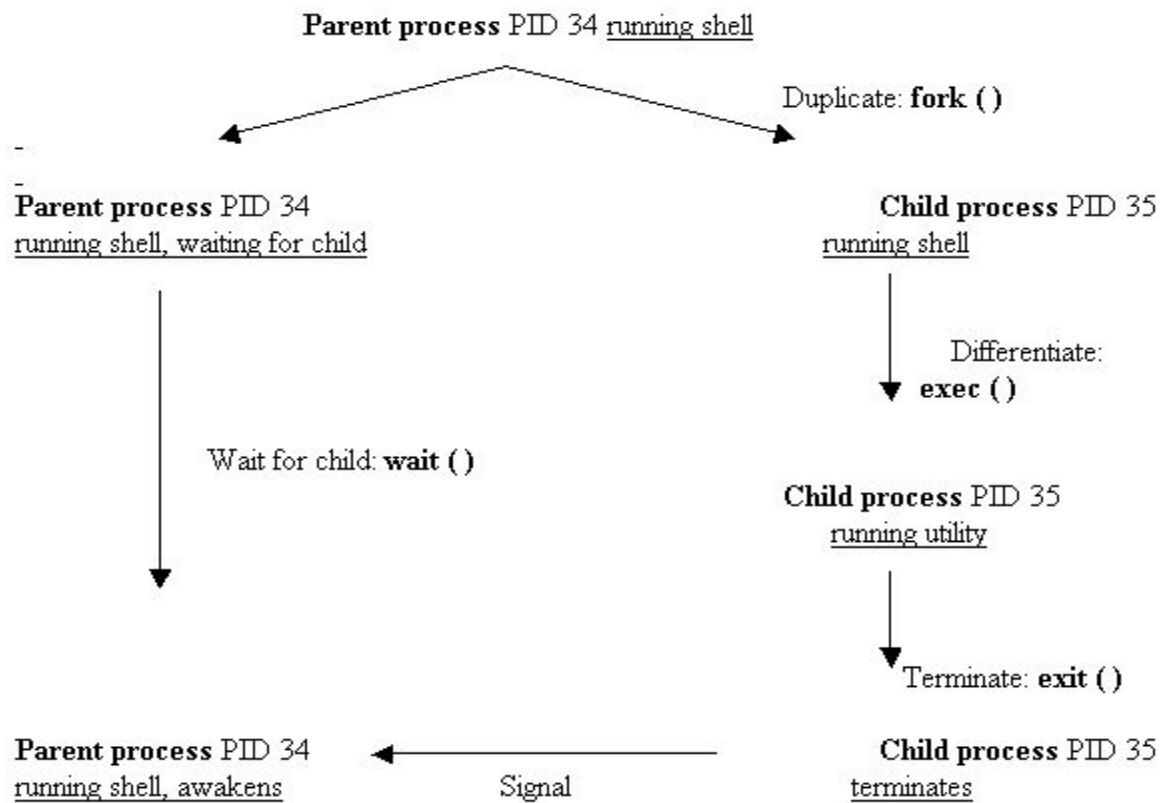
When a child process terminates, its death is communicated to its parent so that the parent may take some appropriate action.

A process that is waiting for its parent to accept its return code is called a **zombie process**.

If a parent dies before its child, the child (orphan process) is automatically adopted by the original “init” process whose PID is 1.

It's very common for a parent process to suspend until one of its children terminates. For example, when a shell executes a utility in the foreground, it duplicates into two shell processes; the child shell process replaces its code with that of utility, whereas the parent shell waits for the child process to terminate. When the child process terminates, the original parent process awakens and presents the user with the next shell prompt.

Here's an illustration of the way that a shell executes a utility:



A program usually runs as a single process. However later we will see how we can make programs run as several separate communicating processes.

Running UNIX commands from C :

We can run commands from a C program just as if they were from the UNIX command line by using the `system()` function.

`int system (char *string)` -- where `string` can be the name of a UNIX utility, an executable shell script or a user program. System returns the exit status of the shell. System is prototyped in `<stdlib.h>`

Example: Call `ls` from a program

File Lab6_0.c :

```
main()  
{  printf(`Files in Directory are:n");  
    system(`ls -l");  
}
```

`system` is a call that is made up of 3 other system calls: `execl()`, `wait()` and `fork()` (which are prototyped in `<unistd.h>`)

Process Creation Concepts :

This section gives an overview of processes and of the steps involved in creating a process and making it run another program.

Each process is named by a process ID number. A unique process ID is allocated to each process when it is created. The **lifetime of a process ends when its termination is reported to its parent process**; at that time, all of the process resources, including its process ID, are freed.

Processes are created with the fork system call (so the operation of creating a new process is sometimes called forking a process). **The child process created by fork is a copy of the original parent process, except that it has its own process ID.**

After forking a child process, both the parent and child processes continue to execute normally.

If you want your program to wait for a child process to finish executing before continuing, you must do this explicitly after the fork operation, by calling `wait`. This function gives you limited information about why the child terminated--for example, its exit status code.

A newly forked child process continues to execute the same program as its parent process, at the point where the fork call returns. You can use the return value from

fork to tell whether the program is running in the **parent** process or the **child**.

Having several processes run the same program is only occasionally useful. **But the child can execute another program using one of the exec functions.** The program that the process is executing is called its process image. **Starting execution of a new program causes the process to forget all about its previous process image; when the new program exits, the process exits too, instead of returning to the previous process image.**

Process Identification :

The **pid_t** data type represents process IDs. You can get the process ID of a process by calling **getpid**. The function **getppid** returns the process ID of the parent of the current process (this is also known as the parent process ID). Your program should include the header files **'unistd.h'** and **'sys/types.h'** to use these functions.

Data Type: pid_t

The **pid_t** data type is a signed integer type which is capable of representing a process ID. In the GNU library, this is an **int**.

Function: pid_t getpid (void)

The **getpid** function returns the **process ID** of the current process.

Function: pid_t getppid (void)

The **getppid** function returns the **process ID of the parent** of the current process.

Creating Multiple Processes :

A special type of process important in the Unix environment is the **daemon**.

The **fork** function is the primitive for creating a process. It is declared in the header file **'unistd.h'**.

Function: pid_t fork (void)

The **fork** function **creates a new process**.

If the operation is **successful**, there are then both **parent** and **child** processes and both see **fork** return, but with different values: it returns a value of **0** in the **child** process and returns the **child's process ID** in the **parent** process.

If process creation **failed**, fork returns a value of **-1** in the **parent** process and **no child is created**.

The specific attributes of the child process that differ from the parent process are:

The child process has its own unique process ID.

The parent process ID of the child process is the process ID of its parent process. The child process gets its own copies of the parent process's open file descriptors.

Subsequently changing attributes of the file descriptors in the parent process won't affect the file descriptors in the child, and vice versa. However, the file position associated with each descriptor is shared by both processes.

The elapsed processor times for the child process are set to zero.

The child doesn't inherit file locks set by the parent process.

The child doesn't inherit alarms set by the parent process.

The set of pending signals for the child process is cleared.

Example Lab6_1.c :

```
tiger> gedit Lab6_1.c
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void) {
    printf("Hello World!\n");
    fork();
    printf("I am after forking\n");
    printf("\tI am process %d.\n", getpid());
}
```

Sample output :

```
Hello World!
I am after forking
    I am process 23848.
I am after forking
    I am process 23847.
```

When this program is executed, it first prints Hello World! . When the fork is executed, an identical process called the child is created. Then both the parent and the child process begin execution at the next statement. Note the following:

When a fork is executed, everything in the parent process is copied to the child process. This includes variable values, code, and file descriptors.

Following the fork, the child and parent processes are completely independent.

There is no guarantee which process will print I am a process first.

The child process begins execution at the statement immediately after the fork, not at the beginning of the program.

A parent process can be distinguished from the child process by examining the return value of the fork call. Fork returns a zero to the child process and the process id of the child process to the parent.

A process can execute as many forks as desired. However, be wary of infinite loops of forks (there is a maximum number of processes allowed for a single user).

Example Lab5_2.c :

Each process prints a message identifying itself.

```
tiger> gedit Lab6_2.c
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
    int pid;
    printf("Hello World!\n");
    printf("I am the parent process and pid is : %d .\n",getpid());
    printf("Here i am before use of forking\n");
    pid = fork( );
    printf("Here I am just after forking\n");
    if (pid == 0)
        printf("I am the child process and pid is :%d.\n",getpid());
    else
        printf("I am the parent process and pid is: %d .\n",getpid());
}
```

Sample Output :

```
Hello World!
I am the parent process and pid is : 23951 .
Here i am before use of forking
Here I am just after forking
I am the child process and pid is :23952.
Here I am just after forking
I am the parent process and pid is: 23951 .
```

Example Lab6_3.c :

Multiple forks:

```
tiger> gedit Lab6_3.c
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
    printf("Here I am just before first forking statement\n");
    fork( );
    printf("Here I am just after first forking statement\n");
    fork( );
}
```

```

printf("Here I am just after second forking statement\n");
fork( );
printf("Here I am just after third forking statement\n");
printf("  Hello World from process %d!\n", getpid( ));
}

```

Sample Output :

```

Here I am just before first forking statement
Here I am just after first forking statement
Here I am just after second forking statement
Here I am just after third forking statement
  Hello World from process 24120!
Here I am just after first forking statement
Here I am just after second forking statement
Here I am just after third forking statement
  Hello World from process 24119!
Here I am just after second forking statement
Here I am just after third forking statement
  Hello World from process 24122!
Here I am just after third forking statement
  Hello World from process 24123!
Here I am just after second forking statement
Here I am just after third forking statement
  Hello World from process 24118!
Here I am just after third forking statement
  Hello World from process 24121!
Here I am just after third forking statement
  Hello World from process 24124!
Here I am just after third forking statement
  Hello World from process 24117!

```

Function: void exit (int status)

exit () terminates the process which calls this function and returns the exit **status** value. Both UNIX and C (forked) programs can read the status value.

By convention, **a status of 0 means normal termination any other value indicates an error or unusual occurrence.** Many standard library calls have errors defined in the sys/stat.h header file. We can easily derive our own conventions.

sleep

A process may **suspend** for a period of time using the **sleep** command

Function: unsigned int sleep (seconds)

Example Lab6_4.c :

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

main ()
{
    int forkresult ;
    printf ("%d: I am the parent. Remember my number!\n", getpid() );
    printf ("%d: I am now going to fork ... \n", getpid() );
    forkresult = fork ( ) ;
    if (forkresult != 0)
    { /* the parent will execute this code */
        printf ("%d: My child's pid is %d\n", getpid (), forkresult ) ;
    }
    else /* forkresult == 0 */
    { /* the child will execute this code */
        printf ("%d: Hi! I am the child.\n", getpid () ) ;
    }
    printf ("%d: like father like son. \n", getpid () ) ;
}
```

Sample Output :

```
28715: I am the parent. Remember my number!
28715: I am now going to fork ...
28716: Hi! I am the child.
28716: like father like son.
28715: My child's pid is 28716
28715: like father like son.
```

Orphan processes :

When a **parent dies before its child**, the child is automatically adopted by the original “**init**” process whose **PID** is **1**. To, illustrate this insert a **sleep** statement into the child’s code. This ensured that the parent process terminated before its child.

Example Lab6_5.c :

```
#include <stdio.h>
```

```

main ( )
{
    int  pid ;

    printf ("I'am the original process with PID %d and PPID %d.\n",
            getpid ( ), getppid ( ) ) ;

    pid = fork ( ) ;    /* Duplicate. Child and parent continue from here */
    if ( pid != 0 )      /* pid is non-zero, so I must be the parent */
    {
        printf ("I'am the parent process with PID %d and PPID %d.\n",
                getpid ( ), getppid ( ) ) ;
        printf ("My child's PID is %d\n", pid ) ;
    }
    else                /* pid is zero, so I must be the child */
    {
        sleep (4) ;      /* make sure that the parent terminates first */
        printf ("I'am the child process with PID %d and PPID %d.\n",
                getpid ( ), getppid ( ) ) ;
    }

    printf ("PID %d terminates.\n", getpid ( ) ) ;
}

```

Sample Output:

```

I'am the original process with PID 5100 and PPID 5011.
I'am the parent process with PID 5100 and PPID 5011.
My child's PID is 5101
PID 5100 terminates.  /* Parent dies */
I'am the child process with PID 5101 and PPID 1.
/* Orphaned, whose parent process is "init" with pid 1 */
PID 5101 terminates.

```

Important points to note:

1. **The Shell acts as the parent process.** All the processes started by the user are treated as the children of **shell**.
2. The **status of a UNIX process** is shown as the **second column** of the process table when viewed by the execution of the **ps** command. Some of the states are:

R: *running*, **O:** *orphan*, **S:** *sleeping*, **Z:** *zombie*.

3. **The child process is given the time slice before the parent process.** This is quite logical. For example, we do not want the process started by us to wait until its parent, which is the UNIX shell finishes. This will explain the order in which the print statement is executed by the parent and the children.

TASKS:

Execute the C programs given in the following problems. **Observe** and **Interpret** the results. You will learn about *child* and *parent* processes, and much more about UNIX processes in general by performing the suggested experiments. UNIX Calls used in the following problems:

getpid(), *getppid()*, *sleep()* and *fork()*.

- 1) Run the following program twice. Both times as a background process, i.e., suffix it with an ampersand "&". Once both processes are running as background processes, view the *process table* using **ps -l** UNIX command. Observe the *process state*, *PID (process ID)* etc. Repeat this experiment to observe the changes, if any. Write your observation about the *Process ID* and *state* of the process.

```
main ( ) {  
    printf ("Process ID is: %d\n", getpid ( ) );  
    printf ("Parent process ID is: %d\n", getppid ( ) );  
    sleep (60);  
    printf ("I am awake. \n");  
}
```

- 2) Run the following program and observe the *number of times* and the *order* in which the print statement is executed. The **fork()** creates a child that is a duplicate of the parent process. The child process begins from the **fork()**. All the statements after the call to **fork()** are executed by the parent process and also by the child process. Draw a family tree of processes and explain the results you observed.

```
main ( ) {  
    fork ( ) ;  
    fork ( ) ;  
    printf ("Parent Process ID is %d\n", getppid ( ) );  
}
```

- 3) Run the following program and observe the result of **time slicing** used by UNIX.

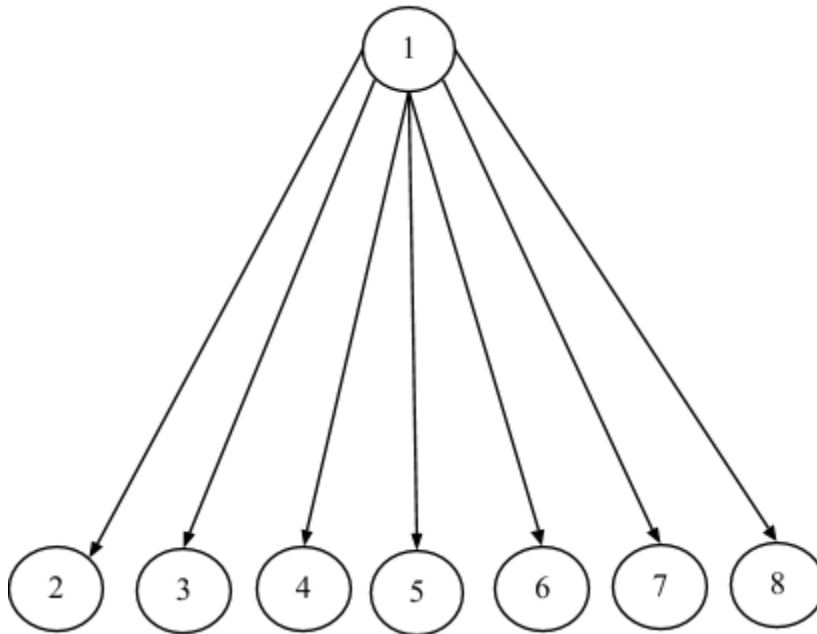
```
main ( ) {  
    int i=0, j=0, pid, k, x ;  
    pid = fork ( ) ;  
    if ( pid == 0 ) {  
        for ( i = 0; i < 20; i++ ) {  
            for (k = 0; k < 10000; k++ );  
        }  
    }  
}
```

```

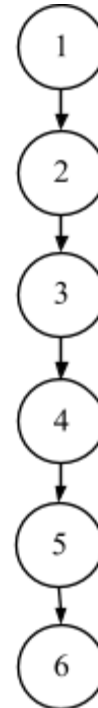
    printf ("Child: %d\n", i) ; } }
else {
    for ( j = 0; j < 20; j++ ){
        for (x = 0; x < 10000; x++ );
        printf ("Parent: %d\n", j) ;    } } }

```

4) Create process fan as shown in figure 1 (a) and fill the figure 1 (a) with actual IDs.



(a)



(b)

Figure 1 Multiple Processes (a) Process Fan (b) Process Chain

5) Create process chain as shown in figure 1(b) and fill the figure 1 (b) with actual IDs.

6) Create process tree as shown in figure 2 and fill the figure 2 with actual IDs.

What are pthreads?

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's.

Pthreads are defined as a set of **C language programming types and procedure calls**. Vendors usually provide a Pthreads implementation in the form of a **header/include file** and a library which you **link** with your program.

Why pthreads?

The primary motivation for using Pthreads is to realize potential program performance gains.

When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication.

Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways: Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads. Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks. Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

Multi-threaded applications will work on a uniprocessor system, yet naturally take advantage of a multiprocessor system, without recompiling.

In a multiprocessor environment, the most important reason for using Pthreads is to take advantage of potential parallelism. This will be the focus of the remainder of this session.

The pthreads API:

The subroutines which comprise the Pthreads API can be informally grouped into three major classes:

Thread management: The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)

Mutexes: The second class of functions deal with a coarse type of synchronization, called a "mutex", which is an abbreviation for "**mutual exclusion**". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

Condition variables: The third class of functions deal with a finer type of synchronization - based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Naming conventions: All identifiers in the threads library begin with **pthread_**

pthread_	Threads themselves and miscellaneous subroutines
pthread_attr	Thread attributes objects
pthread_mutex	Mutexes
pthread_mutexattr	Mutex attributes objects.
pthread_cond	Condition variables
pthread_condattr	Condition attributes objects
pthread_key	Thread-specific data keys

Thread Management Functions:

The function **pthread_create** is used to create a new thread, and the function **pthread_exit** is used by a thread to terminate itself. The function **pthread_join** is used by a thread to wait for termination of another thread.

Function:	int pthread_create (pthread_t *threadhandle, /*Thread handle returned by reference */ pthread_attr_t *attribute, /* Special Attribute for starting thread, may be NULL */ void *(*start_routine)(void *), /* Main Function which thread executes */ void *arg /* An extra argument passed as a pointer */);
Info:	Request the PThread library for creation of a new thread. The return value is 0 on success . The pthread_t is an abstract datatype that is used as a handle to reference the thread.

Function:	Void pthread_exit (void *retval /* return value passed as a pointer */);
-----------	---

Info:	This Function is used by a thread to terminate . The return value is passed as a pointer . This pointer value can be anything so long as it does not exceed the size of (void *). Be careful, this is system dependent. You may wish to return an address of a structure, if the returned data is very large.
-------	---

Function:	Int pthread_join (pthread_t threadhandle, /* Pass threadhandle */ void **returnvalue /* Return value is returned by ref. */);
Info:	Return 0 on success , and negative on failure . The returned value is a pointer returned by reference . If you do not care about the return value, you can pass NULL for the second argument.

Thread Initialization:

Include the pthread.h library :

#include <pthread.h>

Declare a variable of type pthread_t :

pthread_t the_thread

When you compile, add -lpthread to the linker flags :

gcc threads.c -o threads -lpthread

+++++

Initially, threads are created from within a process. Once created, threads are peers, and may create other threads. Note that an "initial thread" exists by default and is the thread which runs main().

+++++

Thread Identifiers:

pthread_self ()

Returns the unique thread ID of the calling thread. The returned data object is opaque can not be easily inspected.

pthread_equal (thread1, thread2)

Compares two thread IDs:

If the two IDs are different 0 is returned, otherwise a non-zero value is returned.

Because thread IDs are opaque objects, the C language equivalence operator == should not be used to compare two thread IDs.

Example: Pthread Creation and Termination:

Lab6_1.c

```

#include <stdio.h>
#include <pthread.h>

void *kidfunc(void *p)
{
    printf ("Kid ID is ---> %d\n", getpid());
}

main ()
{
    pthread_t kid ;
    pthread_create (&kid, NULL, kidfunc, NULL);
    printf ("Parent ID is ---> %d\n", getpid()) ;
    pthread_join (kid, NULL) ;
    printf ("No more kid!\n") ;
}

```

Question: Are the process id numbers of parent and child thread the same or different? Give reason(s) for your answer.

```

=====

```

Lab6_2.c

```

#include <stdio.h>
#include <pthread.h>

int glob_data = 5 ;

void *kidfunc(void *p)
{
    printf ("Kid here. Global data was %d.\n", glob_data) ;
    glob_data = 15 ;
    printf ("Kid Again. Global data was now %d.\n", glob_data) ;
}

main ()
{
    pthread_t kid ;

    pthread_create (&kid, NULL, kidfunc, NULL) ;
    printf ("Parent here. Global data = %d\n", glob_data) ;
    glob_data = 10 ;
    pthread_join (kid, NULL) ;
    printf ("End of program. Global data = %d\n", glob_data) ;
}

```

```
}
```

Question: Do the threads have separate copies of glob_data? Why? Or why not?

.....

Multiple Threads:

The simple example code below creates 5 threads with the **pthread_create()** routine. Each thread prints a "Hello World!" message, and then terminates with a call to **pthread_exit()**.

Lab6_3.c

```
#include <pthread.h>
```

```
#include <stdio.h>
```

```
#define NUM_THREADS 5
```

```
void *PrintHello(void *threadid)
```

```
{
```

```
    printf("\n%d: Hello World!\n", threadid);
```

```
    pthread_exit(NULL);
```

```
}
```

```
int main( )
```

```
{
```

```
    pthread_t  threads [NUM_THREADS];
```

```
    int rc, t;
```

```
    for(t=0; t < NUM_THREADS; t++) {
```

```
        printf ("Creating thread %d\n", t);
```

```
        rc = pthread_create (&threads[t], NULL, PrintHello, (void *) t );
```

```
        if (rc) {
```

```
            printf("ERROR; return code from pthread_create() is %d\n", rc);
```

```
            exit(-1);
```

```
        }
```

```
    }
```

```
    pthread_exit(NULL);
```

```
}
```

Sample output

```
ccse> lab6_3
```

```
Creating thread 0
```

```
Creating thread 1
```

```
Creating thread 2
```

```
Creating thread 3
```

```
Creating thread 4
```

```
0: Hello World!
```

```
1: Hello World!
```

```
2: Hello World!
```

3: Hello World!

4: Hello World!

ccse>

|||||

Difference between process and threads :

Lab6_4.c

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
int this_is_global;
void thread_func( void *ptr );
int main( ) {
int local_main;
int pid, status;
pthread_t thread1, thread2;
printf("First, we create two threads to see better what context they share...\n");
this_is_global=1000;
printf("Set this_is_global=%d\n",this_is_global);
pthread_create( &thread1, NULL, (void*)&thread_func, (void*) NULL);
pthread_create(&thread2, NULL, (void*)&thread_func, (void*) NULL);
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
printf("After threads, this_is_global=%d\n",this_is_global);
printf("\n");
printf("Now that the threads are done, let's call fork..\n");
local_main=17; this_is_global=17;
printf("Before fork(), local_main=%d, this_is_global=%d\n",local_main,
this_is_global);
pid=fork();
if (pid == 0) { /* this is the child */
    printf("In child, pid %d: &global: %X, &local: %X\n", getpid(),
&this_is_global, &local_main);
    local_main=13; this_is_global=23;
    printf("Child set local main=%d, this_is_global=%d\n",local_main,
this_is_global);
    exit(0);
}
else { /* this is parent */
    printf("In parent, pid %d: &global: %X, &local: %X\n", getpid(),
&this_is_global, &local_main);
    wait(&status);
    printf("In parent, local_main=%d, this_is_global=%d\n",local_main,
this_is_global);
}
```

```

}
exit(0);
}
void thread_func(void *dummy) {
int local_thread;
printf("Thread %d, pid %d, addresses: &global: %X, &local: %X\n",
    pthread_self(), getpid(), &this_is_global, &local_thread);
this_is_global++;
printf("In Thread %d, incremented this_is_global=%d\n", pthread_self(),
    this_is_global);
pthread_exit(0);
}

```

Sample output

ccse> lab6_4

First, we create two threads to see better what context they share...

Set this_is_global=1000

Thread 4, pid 2524, addresses: &global: 20EC8, &local: EF20BD6C

In Thread 4, incremented this_is_global=1001

Thread 5, pid 2524, addresses: &global: 20EC8, &local: EF109D6C

In Thread 5, incremented this_is_global=1002

After threads, this_is_global=1002

Now that the threads are done, let's call fork..

Before fork(), local_main=17, this_is_global=17

In child, pid 2525: &global: 20EC8, &local: EFFFFD34

Child set local main=13, this_is_global=23

In parent, pid 2524: &global: 20EC8, &local: EFFFFD34

In parent, local_main=17, this_is_global=17

ccse>

Assignments:

Problem#1:

The following **Box #1** program demonstrates a simple program where the **main thread** creates another thread to print out the numbers from 1 to 20. The **main thread** waits till the child thread finishes.

```

/* Box #1: Simple Child Thread */
#include <pthread.h>
#include <stdio.h>
void *ChildThread(void *argument)
{
    int i;
    for ( i = 1; i <= 20; ++i ){
        printf(" Child Count - %d\n", i);
    }
}

```

```

    }
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t hThread;    int ret;
    ret=pthread_create(&hThread, NULL, (void *)ChildThread, NULL); /* Create
Thread */
    if (ret < 0)
        printf("Thread Creation Failed\n"); return 1;
    pthread_join (hThread, NULL); /* Parent waits for */
    printf("Parent is continuing....\n");
return 0;
}

```

Compile and execute the Box #1 program and show the output and explain why the output is so?

Problem#2:

In the **Box #2** modify the above **Box #1** program such that the main program passes the **count** as argument to the child thread function and the child thread function prints that many **count** print statements.

```
/* Box #2 : Passing Thread Arguments */

#include <pthread.h>
#include <stdio.h>
void *ChildThread (int argument){
    int i;
    .....
    pthread_exit(NULL);
}
int main(void){
    pthread_t hThread;
    pthread_create (.....);
    pthread_join (hThread, NULL);
    printf ("Parent is continuing....\n");    return 0;
}
```

Compile and Execute the **Box #2** program and show the output and explain why is the output so?

Problem#3:

Write a program **Box #3** by removing **pthread_exit** function from child thread function and check the output? Is it the same as output of **Box #2**? If so Why? Explain?

```
/* Box #3: Implicit Thread Exit */
#include <pthread.h>
#include <stdio.h>
void ChildThread (int argument){
    int i;
    .....
    /* No pthread_exit function */
}
int main(void){
    pthread_t hThread;
    pthread_create (.....);
    pthread_join (hThread, NULL);
    printf ("Parent is continuing....\n");    return 0;
}
```


Operating Systems

LAB 8

Threads Synchronization

Objective:

When multiple threads are running they will invariably need to communicate with each other in order **synchronize** their execution. One main benefit of using threads is the ease of using synchronization facilities.

Threads need to synchronize their activities to effectively interact. This includes:

Implicit communication through the modification of shared data

Explicit communication by informing each other of events that have occurred.

This lab describes the synchronization types available with threads and discusses when and how to use synchronization.

Why we need Synchronization and how to achieve it:

Suppose the multiple threads share the **common address space** (thru a common variable), then there is a problem.

THREAD A

```
x = common_variable ;  
x++ ;  
common_variable = x ;
```

THREAD B

```
y = common_variable ;  
y-- ;  
common_variable = y ;
```

If threads execute this code independently it will lead to garbage. The access to the **common_variable** by both of them simultaneously is prevented by having a lock, performing the thing and then releasing the lock.

Mutual exclusion and Race Conditions:

Mutual exclusion can prevent data inconsistencies due to **race conditions**.

A **race condition** often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed.

Consider, for example, a single counter, **X**, that is incremented by two threads, **A** and **B**. If **X** is originally **1**, then by the time threads **A** and **B** increment the counter, **X** should be **3**. Both threads are independent entities and have no synchronization between them. Although the C statement **X++** looks simple enough to be atomic, the generated assembly code may not be, as shown in the following pseudo-assembler code:

```

move    X,    REG
inc     REG
move    REG, X

```

If both threads are executed concurrently on two CPUs, or if the scheduling makes the threads alternatively execute on each instruction, the following steps may occur:

Situation 1:

Thread **A** executes the first instruction and puts **X**, which is **1**, into the thread **A** register. Then thread **B** executes and puts **X**, which is **1**, into the thread **B** register. The following illustrates the resulting registers and the contents of memory **X**.

Thread A Register	Thread B Register	Memory X
1	1	1

Situation 2:

Next, thread **A** executes the second instruction and increments the content of its register to **2**. Then thread **B** increments its register to **2**. Nothing is moved to memory **X**, so memory **X** stays the same. The following illustrates the resulting registers and the contents of memory **X**.

Thread A Register	Thread B Register	Memory X
2	2	1

Situation 3:

Last, thread **A** moves the content of its register, which is now **2**, into memory **X**. Then thread **B** moves the content of its register, which is also **2**, into memory **X**, overwriting thread **A**'s value. The following illustrates the resulting registers and the contents of memory **X**.

Thread A Register	Thread B Register	Memory X
2	2	2

Note that in most cases thread **A** and thread **B** will execute the three instructions one after the other, and the result would be **3**, as expected. Race conditions are usually difficult to discover, because they occur intermittently.

To avoid this race condition, each thread should lock the data before accessing the counter and updating memory **X**. For example, if thread **A** takes a lock and updates the counter, it leaves memory **X** with a value of **2**. Once thread **A** releases the lock, thread **B** takes the lock and updates the counter, taking **2** as its initial value for **X** and incrementing it to **3**, the expected result.

Task 1: Implement producer/consumer problem using threads.

Task 2: Solve the critical section problem in task1 using:

- i. **Peterson solution**
- ii. **Test and set instruction**

Operating Systems LAB 9

Inter-Process Communication - Pipes

Objective:

- To learn and practice how processes communicate among themselves
- To use Pipes and signals
- Practice the following system calls

pipe
dup / dup2

Introduction:

Now that we know how to create processes, let us turn our attention to make the processes communicate among themselves. There are many mechanisms through which the processes communicate and in this lab we will discuss two such mechanisms: **Pipes** and **Signals**. A **pipe** is used for one-way communication of a stream of bytes. **Signals** inform processes of the occurrence of asynchronous events. In this lab we will learn how to create pipes and how processes communicate by reading or writing to the pipe and also how to have a two-way communication between processes. This lab also discusses how the default signals handlers can be replaced by user-defined handlers for particular signals and also how the processes can ignore the signals.

By learning about signals, you can "protect" your programs from Control-C, arrange for an alarm clock signal to terminate your program if it takes too long to perform a task, and learn how UNIX uses signals during everyday operations.

Pipes

Pipes are familiar to most Unix users as a shell facility. For instance, to print a sorted list of **who** is logged on, you can enter this command line:

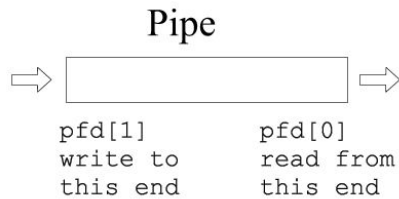
who | sort | lpr

There are **three processes** here, connected with **two pipes**. **Data flows in one direction only**, from **who** to **sort** to **lpr**. It is also possible to set up **bidirectional pipelines** (from process A to B, and from B back to A) and **pipelines in a ring** (from A to B to C to A) using system calls. The shell, however, provides no notation for these more elaborate arrangements, so they are unknown to most Unix users.

We'll begin by showing some simple examples of processes connected by a **one-directional pipeline**.

pipe System Call

```
int pfd[2];  
int pipe (pfd); /* Returns 0 on success or -1 on error */
```



I/O with a pipe

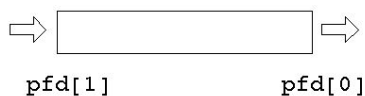
These two file descriptors can be used for block I/O

```
write(pfd[1], buf, SIZE);
read(pfd[0], buf, SIZE);
```

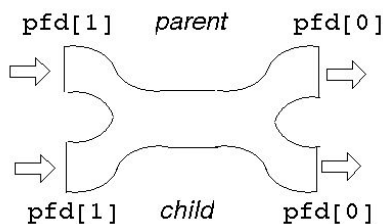
Fork and a pipe

A single process would not use a pipe. They are used when two processes wish to communicate in a one-way fashion. A process splits in two using `fork ()`. A pipe opened before the fork becomes shared between the two processes.

Before fork



After fork



This gives *two read ends* and *two write ends*. The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed.

Either process can write into the pipe, and either can read from it. Which process will get what is not known.

For predictable behavior, *one of the processes must close its read end, and the other must close its write end*. Then it will become a simple pipeline again.

Suppose the parent wants to write down a pipeline to a child. *The parent closes its read end, and writes into the other end. The child closes its write end and reads from the other end. When the processes have ceased communication, the parent closes its write end. This means that the child gets eof on its next read, and it can close its read end.*

Pipes use the buffer cache just as ordinary files do. Therefore, the benefits of writing and reading pipes in units of a block (usually 512 bytes) are just as great. A single **write**

execution is atomic, so if 512 bytes are written with a single system call, the corresponding **read** will return with 512 bytes (if it requests that many). It will not return with less than the full block. However, if the writer is not writing complete blocks, but the reader is trying to read complete blocks, the reader may keep getting partial blocks anyhow. This won't happen if the writer is faster than the reader, since then the writer will be able to fill the pipe with a complete block before the reader gets around to reading anything. Still, it's best to buffer writes and reads on pipes, and this is what the Standard I/O Library does automatically.

```
#include <stdio.h>
#define SIZE 1024
main() {
    int pfd[2];
    int nread;
    int pid;
    char buf[SIZE];
    if (pipe(pfd) == -1) {
        perror("pipe failed");
        exit(1);
    }
    if ((pid = fork()) < 0) {
        perror("fork failed");
        exit(2);
    }
    if (pid == 0) {
        /* child */
        close(pfd[1]);
        while ((nread = read(pfd[0], buf, SIZE)) != 0)
            printf("child read %s\n", buf);
        close(pfd[0]);
    }
    else {
        /* parent */
        close(pfd[0]);
        strcpy(buf, "hello...");
        /* include null terminator in write */
        write(pfd[1], buf, strlen(buf)+1);
        close(pfd[1]);
    }
}
```

Given that **we have two processes, how can we connect them so that one can read from a pipe what the other writes? We can't.** Once the processes are created they can't be connected, because there's **no way for the process that makes the pipe to pass a file descriptor to the other process.** It can pass the file descriptor number, of course, but that number won't be valid in the other process. **But if we make a pipe in one process before creating the other process, it will inherit the pipe file descriptors, and they**

will be valid in both processes. Thus, two processes communicating over a pipe can be parent and child, or two children, or grandparent and grandchild, and so on, but they must be related, and the pipe must be passed on at birth. In practice, this may be a severe limitation, because if a process dies there's no way to recreate it and reconnect it to its pipes -- the survivors must be killed too, and then the whole family has to be recreated.

In general, then, here is how to connect two processes with a pipe:

1. Make the pipe.
2. Fork to create the reading child.
3. In the child close the writing end of the pipe, and do any other preparations that are needed.
4. In the child execute the child program.
5. In the parent close the reading end of the pipe.
6. If a second child is to write on the pipe, create it, make the necessary preparations, and execute its program. If the parent is to write, go ahead and write.

Here's a small program that uses a pipe to allow the parent to read a message from its child:

```
#include <stdio.h>
#include <string.h>
#define READ 0
#define WRITE 1
char* phrase = "This is ICS431 lab time" ;
main () {
    int  fd [2], bytesread ;
    char message [100] ;
    pipe ( fd ) ;
    if ( fork ( ) == 0 ) {                /* child, writer */
        close ( fd [READ] ) ;            /* close unused end */
        write ( fd [WRITE], phrase, strlen (phrase) + 1 ) ;
        close ( fd [WRITE] ) ;           /* close used end */
    }
    else {                                /* parent, reader */
        close ( fd [WRITE] ) ;           /* close unused end */
        bytesread = read (fd [READ], message, 100) ;
        printf ("Read %d bytes : %s\n", bytesread, message) ;
        close ( fd [READ] ) ;            /* close used end */
    }
}
```

Implementation of Redirection

When a process **forks**, the child inherits a copy of its parent's file descriptors. When a process **execs**, the standard input, output, and error channels remain unaffected. The UNIX shells uses these two pieces of information to implement redirection.

To perform redirection, the shell performs the following series of actions:

- The parent shell forks and then waits for the child shell to terminate.
- The child shell opens the file "output", creating it or truncating as necessary.
- The child shell then duplicates the file descriptor of "output" to the standard output file descriptor, **number 1**, and then closes the original descriptor of "output". All standard output is therefore redirected to "output".
- The child shell then **exec's** the **ls** utility. Since the file descriptors are inherited during an **exec ()**, all of standard output of **ls** goes to "output".
- When the child shell terminates, the parent resumes. The parent's file descriptors are unaffected by the child's actions, as each process maintains its own private descriptor table.

```
#include <stdio.h>
#include <sys/file.h>
main (argc, argv)
int  argc ;
char *argv[ ] ;
{
    int  fd ;    /* file descriptor or pointer */
    fd = open (argv[1], O_CREAT | O_TRUNC | O_RDWR, 0777) ;
           /* open file named in argv[1] */
    dup2 (fd, 1) ;    /* and assign it to fd file pointer */
    close (fd) ; /* duplicate fd with 1 which is standard output (the monitor) */
    execvp (argv[2], &argv[2]) ;
    /* the output is not printed on screen but is redirected to "output" file */
    printf ("End\n") ; /* should never execute */
}
```

dup / dup2 System Call

```
int dup (int oldfd )
int dup2 (int oldfd, int newfd )
```

dup () finds the smallest free file descriptor entry and points it to the same file as *oldfd*. **dup2 ()** closes *newfd* if it's currently active and then points it to the same file as *oldfd*. In both cases, the **original and copied file descriptors share the same file pointer and access mode**.

They both return the **index of the new file descriptor** if **successful**, and **-1** otherwise.

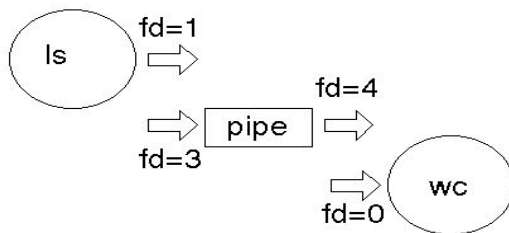
dup/dup2 duplicates an existing file descriptor, giving a new file descriptor that is open to the same file or pipe. The two share the same file pointer, just as an inherited file descriptor shares the file pointer with the corresponding file descriptor in the parent. The call fails if the argument is bad (not open) or if 20 file descriptors are already open.

A pipeline works because the two processes know the file descriptor of each end of the pipe. Each process has a stdin (0), a stdout (1) and a stderr (2). The file descriptors will depend on which other files have been opened, but could be 3 and 4, say.

Suppose one of the processes replaces itself by an "exec". The new process will have files for descriptors 0, 1, 2, 3 and 4 open. How will it know which are the ones belonging to the pipe? It can't.

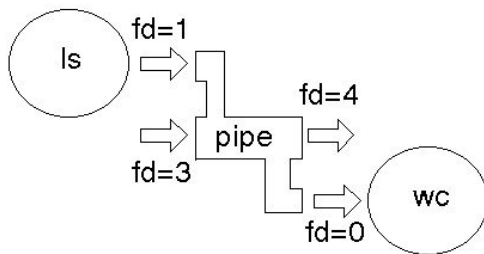
Example:

To implement "ls | wc" the shell will have created a pipe and then forked. The parent will exec to be replaced by "ls", and the child will exec to be replaced by "wc". The write end of the pipe may be descriptor 3 and the read end may be descriptor 4. "ls" normally writes to 1 and "wc" normally reads from 0. How do these get matched up?

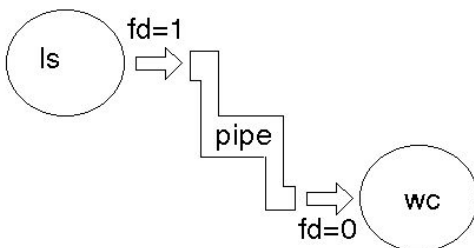


The **dup/dup2** function call takes an existing file descriptor, and another one that it "would like to be". Here, fd=3 would also like to be 1, and fd=4 would like to be 0. So we **dup** fd=3 as 1, and **dup** fd=4 as 0. Then the old fd=3 and fd=4 can be closed as they are no longer needed.

After dup



After close



The UNIX shells use unnamed pipes to build pipelines. They use a trick similar to the redirection mechanism to connect the standard output of one process to standard input of another. To illustrate this approach, here's the program that executes two named programs, connecting the standard output of the first to the standard input of the second.

```
#include <stdio.h>
#include <string.h>

#define READ  0
#define WRITE 1

main (argc, argv)
int  argc ;
char* argv[] ;
{
    int  pid, fd [2] ;
    if (pipe(fd) == -1)
    {
        perror("pipe failed");
        exit(1);
    }
    if ((pid = fork( )) < 0)
    {
        perror("fork failed");
        exit(2);
    }
    if ( pid != 0 )                /* parent, writer */
    {
        close ( fd [READ] ) ;      /* close unused end */
        dup2 ( fd [WRITE], 1 ) ;   /* duplicate used end to standard out */
        close ( fd [WRITE] ) ;     /* close used end */
        execlp ( argv[1], argv[1], NULL ) ; /* execute writer program */
    }
    else                          /* child, reader */
    {
        close ( fd [WRITE] ) ;     /* close unused end */
        dup2 ( fd [READ], 0 ) ;    /* duplicate used end to standard input */
        close ( fd [READ] ) ;      /* close used end */
        execlp ( argv[2], argv[2], NULL ) ; /* execute reader program */
    }
}
```

Run the above program as

a.out who wc

Variations

Some common variations on this method of IPC are:

1. A pipeline may consist of three or more process (such as a C version of `ps | sed 1d | wc -l`). In this case there are lots of choices
 - i. The parent can fork twice to give two children.
 - ii. The parent can fork once and the child can fork once, giving a parent, child and grandchild.
 - iii. The parent can create two pipes before any forking. After a fork there will then be a total of 8 ends open (2 processes * two ends * 2 pipes). Most of these will have to be closed to ensure that there ends up only one read and only one write end.
 - iv. As many ends as possible of a pipe may be closed before a fork. This minimizes the number of closes that have to be done after forking.
2. A process may want to both write to and read from a child. In this case it creates two pipes. One of these is used by the parent for writing and by the child for reading. The other is used by the child for writing and the parent for reading.

Assignments:

1) Create your own **pipe-redirect** command so that the output of given command goes to a given file.

Eg. `./a.out ls file1`

The output of `ls` command is stored in `file1`.

Operating Systems
LAB 10
Shared Memory Management (IPC using Shared Memory)

Objective:

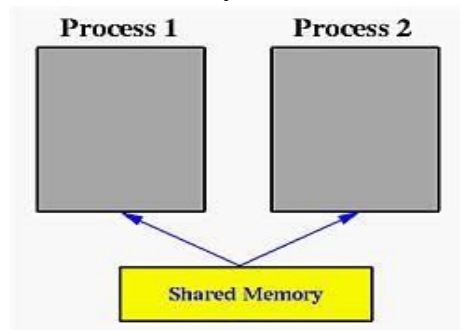
The aim of this laboratory is to show you how the processes can communicate among themselves using the Shared Memory regions. *Shared Memory* is an efficient means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access. A shared segment can be attached multiple times by the same process. **A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory.** In this lab the following issues related to shared memory utilization are discussed:

- **Creating a Shared Memory Segment**
- **Controlling a Shared Memory Segment**
- **Attaching and Detaching a Shared Memory Segment**

Introduction:

What is Shared Memory?

In the discussion of the `fork()` system call, we mentioned that a parent and its children have **separate address spaces**. While this would provide a more secured way of executing parent and children processes (because they will not interfere each other), they shared nothing and have no way to communicate with each other. **A shared memory is an extra piece of memory that is attached to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it.** Consequently, race conditions may occur if memory accesses are not handled properly. The following figure shows two processes and their address spaces. The yellow rectangle is a **shared memory** attached to both address spaces and **both process 1 and process 2 can have access to this shared memory as if the shared memory is part of its own address space.** In some sense, the original address spaces is "extended" by attaching this shared memory.



Shared memory is a feature supported by UNIX System V, including Linux, SunOS and Solaris. One process must explicitly ask for an area, using a key, to be shared by other processes. This process will be called the server. All other processes, the clients, that know the shared area can access it. However, there is no protection to a shared memory and

any process that knows it can access it freely. To protect a shared memory from being

accessed at the same time by several processes, a synchronization protocol must be setup.

A shared memory segment is identified by a unique integer, the shared memory ID. The shared memory itself is described by a structure of type **shmid_ds** in header file **sys/shm.h**. To use this file, files **sys/types.h** and **sys/ipc.h** must be included. Therefore, your program should start with the following lines:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

A general scheme of using shared memory is the following:

For a server, it should be started before any client. The **server** should perform the following tasks:

1. Ask for a shared memory with a memory key and memorize the returned shared memory ID. This is performed by system call **shmget()**.
2. Attach this shared memory to the server's address space with system call **shmat()**.
3. Initialize the shared memory, if necessary.
4. Do something and wait for all clients' completion.
5. Detach the shared memory with system call **shmdt()**.
6. Remove the shared memory with system call **shmctl()**.

For the **client** part, the procedure is almost the same:

1. Ask for a shared memory with the same memory key and memorize the returned shared memory ID.
2. Attach this shared memory to the client's address space.
3. Use the memory.
4. Detach all shared memory segments, if necessary.
5. Exit.

Asking for a Shared Memory Segment - **shmget()**

The system call that requests a shared memory segment is **shmget()**. It is defined as follows:

```
shm_id = shmget ( key_t k,      /* the key for the segment */
                  int size,    /* the size of the segment */
                  int flag );  /* create/use flag */
```

In the above definition, **k** is of type **key_t** or **IPC_PRIVATE**. It is the numeric key to be assigned to the returned shared memory segment. **size** is the size of the requested shared memory. The purpose of **flag** is to specify the way that the shared memory will be used. For our purpose, only the following two values are important:

1. **IPC_CREAT | 0666** for a **server** (i.e., **creating and granting read and write access to the server**)
2. **0666** for any **client** (i.e., **granting read and write access to the client**)

Note that due to Unix's tradition, **IPC_CREAT** is correct and **IPC_CREATE** is not!!! If **shmget()** can successfully get the requested shared memory, its function value is a **non-negative integer**, the **shared memory ID**; otherwise, the function value is **negative**. The following is a **server** example of requesting a **private shared memory of four integers**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

.....
int  shm_id; /* shared memory ID */
.....

shm_id = shmget (IPC_PRIVATE, 4*sizeof(int), IPC_CREAT | 0666);
if (shm_id < 0) {
    printf("shmget error\n");
    exit(1);
}
```

/* now the shared memory ID is stored in **shm_id** */

If a client wants to use a shared memory created with **IPC_PRIVATE**, it must be a child process of the server, created after the parent has obtained the shared memory, so that the private key value can be passed to the child when it is created. For a client, changing **IPC_CREAT | 0666** to **0666** works fine. **A warning to novice C programmers: don't change 0666 to 666. The leading 0 of an integer indicates that the integer is an octal number.** Thus, 0666 is 110110110 in binary. If the leading zero is removed, the integer becomes six hundred sixty six with a binary representation 1111011010.

Server and clients can have a parent/client relationship or run as separate and unrelated processes. In the former case, if a shared memory is requested and attached prior to forking the child client process, then the server may want to use **IPC_PRIVATE** since the child receives an identical copy of the server's address space which includes the attached shared memory. However, if the server and clients are separate processes, using **IPC_PRIVATE** is unwise since the clients will not be able to request the same shared memory segment with a unique and unknown key.

Keys

Unix requires a **key** of type **key_t** defined in file **sys/types.h** for requesting resources such as shared memory segments, message queues and semaphores. **A key is simply an integer of type key_t; however, you should not use int or long, since the length of a key is system dependent.**

There are three different ways of using keys, namely:

1. a specific integer value (e.g., 123456)
2. a key generated with function `ftok()`
3. a uniquely generated key using `IPC_PRIVATE` (i.e., a private key).

The first way is the easiest one; however, its use may be very risky since a process can access your resource as long as it uses the same key value to request that resource. The following example assigns 1234 to a key:

```
key_t SomeKey;  
SomeKey = 1234;
```

The `ftok()` function has the following prototype:

```
key_t ftok (  
    const char *path, /* a path string */  
    int id           /* an integer value */  
);
```

Function `ftok()` takes a character string that identifies a path and an integer (usually a character) value, and generates an integer of type `key_t` based on the first argument with the value of `id` in the most significant position. For example, if the generated integer is 35028A5D16 and the value of `id` is 'a' (ASCII value = 6116), then `ftok()` returns 61028A5D16. That is, 6116 replaces the first byte of 35028A5D16, generating 61028A5D16.

Thus, as long as processes use the same arguments to call `ftok()`, the returned key value will always be the same. The most commonly used value for the first argument is `"."`, the current directory. If all related processes are stored in the same directory, the following call to `ftok()` will generate the same key value:

```
#include <sys/types.h>  
#include <sys/ipc.h>  
key_t SomeKey;  
SomeKey = ftok(".", 'x');
```

After obtaining a key value, it can be used in any place where a key is required. Moreover, the place where a key is required accepts a special parameter, `IPC_PRIVATE`. In this case, the system will generate a unique key and guarantee that no other process will have the same key. If a resource is requested with `IPC_PRIVATE` in a place where a key is required, that process will receive a unique key for that resource. Since that resource is identified with a unique key unknown to the outsiders, other processes will not be able to share that resource and, as a result, the requesting process is guaranteed that it owns and accesses that resource exclusively.

Attaching a Shared Memory Segment to an Address Space

- `shmat()`

Why to attach?

Creating shared memory space is not good enough. You asked the OS for it, but when you want access to the shared memory, the process needs to make it look like a well known data structure. In IPCS terminology this is *attaching the shared memory*. Basically, you are asking the OS to change the relationship of the created shared memory to a data structure mapped into your address space. In particular treating it as a particular structure. *shmget* creates shared memory and *shmat* maps shared memory into a process' address space. Depending on the order of actions shared memory can be created and attached before any *forks*, and thus since children inherit, children will automatically have the shared memory mapped to a common data structure.

Attaching

Suppose process 1, a server, uses **shmget()** to request a shared memory segment successfully. That shared memory segment exists somewhere in the memory, but is not yet part of the address space of process 1. Similarly, if process 2 requests the same shared memory segment with the same key value, process 2 will be granted the right to use the shared memory segment; but it is not yet part of the address space of process 2. To make a requested shared memory segment part of the address space of a process, use **shmat()**.

After a shared memory ID is returned, the next step is to **attach it to the address space of a process**. This is done with system call **shmat()**. The use of **shmat()** is as follows:

```
shm_ptr = shmat (
                int    shm_id,    /* shared memory ID */
                char   *ptr,      /* a character pointer */
                int     flag );   /* access flag */
```

System call **shmat()** accepts a shared memory ID, **shm_id**, and attaches the indicated shared memory to the program's address space. The returned value is a pointer of type (void *) to the attached shared memory. Thus, casting is usually necessary. If this call is **unsuccessful**, the return value is **-1**. Normally, the second parameter is **NULL**. If the flag is **SHM_RDONLY**, this shared memory is attached as a **read-only memory**; otherwise, it is **readable** and **writable**.

In the following **server's** program, it **asks for and attaches a shared memory of four integers**.

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

```

#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

...
int    shm_id; key_t mem_key; int    *shm_ptr;
mem_key = ftok(".", 'a');
shm_id = shmget(mem_key, 4*sizeof(int), IPC_CREAT | 0666);
if (shm_id < 0)
{
    printf("*** shmget error (server) ***\n");
    exit(1);
}

shm_ptr = (int *) shmat(shm_id, NULL, 0); /* attach */
if ((int) shm_ptr == -1) {
    printf("*** shmat error (server) ***\n");
    exit(1);
}

```

The following is the counterpart of a **client**.

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

...
int    shm_id;
key_t mem_key;
int    *shm_ptr;
mem_key = ftok(".", 'a');
shm_id = shmget(mem_key, 4*sizeof(int), 0666);
if (shm_id < 0) {
    printf("*** shmget error (client) ***\n");
    exit(1);
}

shm_ptr = (int *) shmat(shm_id, NULL, 0);
if ((int) shm_ptr == -1)
{ /* attach */
    printf("*** shmat error (client) ***\n");
    exit(1);
}

```

Note that the above code assumes the **server** and **client** programs are in the **current directory**. In order for the client to run correctly, the server must be started first and the client can only be started after the server has successfully obtained the shared memory.

Suppose process 1 and process 2 have successfully attached the shared memory segment. This shared memory segment will be part of their address space, although the **actual address could be different** (i.e., the starting address of this shared memory segment in the address space of process 1 may be different from the starting address in the address space of process 2).

Detaching and Removing a Shared Memory Segment

- **shmdt()** and **shmctl()**

System call **shmdt()** is used to **detach a shared memory**. After a shared memory is detached, it cannot be used. However, it is still there and **can be re-attached back to a process's address space, perhaps at a different address**. To **remove a shared memory**, use **shmctl()**.

The only argument to **shmdt()** is the shared memory address returned by **shmat()**. Thus, the following code detaches the shared memory from a program:

```
shmdt(shm_ptr);
```

where **shm_ptr** is the **pointer to the shared memory**. This pointer is returned by **shmat()** when the shared memory is attached. If the detach operation fails, the returned function value is **non-zero**.

To **remove a shared memory segment**, use the following code:

```
shmctl(shm_id, IPC_RMID, NULL);
```

where **shm_id** is the shared memory ID. **IPC_RMID** indicates this is a remove operation. Note that after the removal of a shared memory segment, if you want to use it again, you should use **shmget()** followed by **shmat()**.

Example Programs

Two different processes communicating via shared memory

We develop two programs here that illustrate the passing of a simple piece of memory (a string) between the processes if **running simultaneously**:

shm_server.c

- simply creates the string and shared memory portion.
- run it in background

shm_client.c

- attaches itself to the created shared memory portion and prints the string.

Parent and Child processes communicating via shared memory

parent_child.c

- One parent places characters in shared memory, and child reads it.

Assignment:

Inter Process Communication between three processes using Shared Memory

Write a program that creates a shared memory segment and waits until two other separate processes writes something into that shared memory segment after which it prints what is

written in shared memory. For the communication between the processes to take place assume that the **process 1** writes **1** in first position of shared memory and waits; **process 2** writes **2** in first position of shared memory and goes on to write '**hello**' and then **process 3** writes **3** in first position of shared memory and goes on to write '**memory**' and finally the **process 1** prints what is in shared memory written by two other processes.

Note:

Name the three (above said) programs as **process1.c**, **process2.c** and **process3.c**
Compile them and name the executable files as **p1**, **p2** and **p3**

The output should be something like this:

```
tiger > ./p1 &  
[1] 6720  
tiger > Process1:- I have put the message 1  
tiger > ./p2  
tiger > Process1:- Process2 has put the message 2 hello  
tiger > ./p3  
tiger > Process1:- Process3 has put the message 3 memory  
Process1:- I am quitting  
[1] Done ./p1  
tiger >
```

In the above output **1, 2 hello and 3 memory** should be printed by reading from the shared memory.

Operating Systems LAB 11

Simulation of Non-preemptive Process Scheduling Algorithms

WRITE A C PROGRAM FOR CPU SCHEDULING ALGORITHM FOR FCFS

DESCRIPTION:

The implementation of FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked on to the tail of the queue. When the CPU is free it is allocated to the process at that end of the queue. The running process is then removed from the queue.

Data Structures Required:

For all CPU scheduling algorithm implementations, we need to have the following arrays.

A two-dimensional array "process [20][1 0]" of data type float.

This array stores the job numbers, their arrival times, and their burst times, which are read from the user.

And this array also stores the additional data after some calculations. They are...

1. Start time of the job,
2. Finish time of the job,
3. Waiting time of job,
4. Turn around time of job. .

A row of the process array contains the following:

Process[1,0] = job number,
Process[1,1] = arrival time,
Process[1,2] = burst time, "

And after calculations...

Process[1,3] = start time,
Process[1,4] = finish time,
Process[1,5] = waiting time,
Process[1 ,6] = turn around time,
Remaining columns may be used while implementing the other algorithms.

Variables avgwt, avgtat, of float indicating average waiting time and average turnaround time respectively, and variable TOT_JOBS for keeping track of number of jobs.

Functions Needed:

While implementing algorithms one can implement his/her own logic and can write own functions, which is up to the programmers. As far as possible avoid using global variables.

Though it is so, the suggested functions for implementing these algorithms are

1. for reading the job entries (read_job_entry())
2. FCFS() for fcfs algorithm
3. SJF_non_preempt()for SJF algorithm,
4. Calculate() for doing various calculations like start times, finish times, waiting times, turn around times of jobs.
5. Sorting() for sorting the job entries by their arrival time (for FCFS), or by their burst time (for SJF), or by their priority (for Priority algo).
6. Print_sorted() for printing inputted data after sorting.
7. Printing(): printing the outputs after all the calculations are over.

PROCEDURE FOR FCFS

Read the number of jobs into variable tot_obs.

Then read the data for jobs as:

Job number, into process [1, 0],

Job arrival time into process [1, 1],

Job burst into process [1, 2], through read_job_entry().

Then sort the job entries by their arrival times because FCFS algorithm works by arrival times.

Print the sorted jobs on the output.

Then go for calculations.

Then Print the results on the output.

The read function can be called from the main function. Then FCFS function can be as

Function:

```
FCFS() {  
    sort(...);  
    printf("The      Scheduling  
    according to FCFS:");  
    print_sorted ( . . .);  
    calculated(...);  
    getch( );  
    printing( );  
    getch( );  
}
```

Output and menus:

Program for Process <u>Scheduling.</u> Press I for FCFS 2 SJF Non Preempt 3 Exit Default Input again

Output:

Enter the Total number of jobs:

Enter Job No:

Enter Arrival time:

Enter Burst time:

.
.
.

Up to -total number of jobs these information is read from; inputs. After these inputs are over a menu should be displayed which is shown later in this. Then print the sorted jobs as below.

The sorted Jobs are

<u>Job No</u>	<u>Job .Arrival Time</u>	<u>Job Burst Time</u>
...
...

Then scheduling output:

Scheduling According to FCFS is

Job No.	AT	BT	ST	FT	WT	TAT
...
...

WRITE A C PROGRAM FOR CPU SCHEDULING ALGORITHM FOR SJF

DESCRIPTION:

The SJF algorithm also can be implemented as FCFS, but on this the jobs are sorted by their burst time. A job with shortest burst time will be scheduled first i.e. sorting should be done from short to large time of burst time. Take arrival time also in consideration. This is SJF Non pre-emptive algorithm. While implementing Priority scheduling we have read the priority also and they should be sorted by highest priority to lowest priority.

It maintains the Ready queue in order of increasing job lengths. When a job comes in, insert it in the ready queue based on its length. When current process is done, pick the one at the head of the queue and run it.

Data Structures Required:

For all CPU scheduling algorithm implementations, we need to have the following arrays.

A two-dimensional array "process [20][1 0]" of data type float.

This array stores the job numbers, their arrival times, and their burst times, which are read from the user.

And this array also stores the additional data after some calculations. They are...

1. start time of the job,
2. finish time of the job,
3. waiting time of job,
4. Turn around time of job. .

A row of the process array contains the-following: '

Process[1,0] = job number,

Process[1,1] = arrival time,

Process[1,2] = burst time, "

And after calculations...

Process[1,3] = start time,

Process[1,4] = finish time,

Process[1,5] = waiting time,

Process[1 ,6] = turn around time,

Remaining columns may be used while implementing the other algorithms.

Variables avgwt, avgtat, of float indicating average waiting time and average turn around time respectively, and variable TOT_JOBS for keeping track of number of jobs.

Operating Systems LAB 12

Simulation of Preemptive Process Scheduling Algorithms

IMPLEMENT THE PREEMPTIVE PRIORITY SCHEDULING ALGORITHM. TAKE ARRIVAL TIME INTO CONSIDERATION.

DESCRIPTION:

Run highest-priority processes first. Re-insert process in run queue behind all processes of greater or equal priority if a high priority process preempts the current process.

- Allows CPU to be given preferentially to important processes.
- Scheduler adjusts dispatcher priorities to achieve the desired overall priorities for the processes, e.g. one process gets 90% of the CPU.

Comments: In priority scheduling, processes are allocated to the CPU on the basis of an externally assigned priority.

Problem: Priority scheduling may cause low-priority processes to starve

Solution: (AGING) starvation can be compensated if the priorities are internally computed. Suppose one parameter in the priority assignment function is the amount of time the process has been waiting. The longer a process waits, the higher its priority becomes. This strategy tends to eliminate the starvation problem.

Waiting Time: It is the sum of the periods spent waiting in the ready queue.

Turnaround Time: The interval from the time of submission of a process to the time of completion is the turnaround time.

Response Time: It is the amount of time takes to start responding, but not the time that it takes to output that response.

Throughput: The number of processes that are completed per unit called throughput.

IMPLEMENT THE ROUND-ROBIN SCHEDULING ALGORITHM.

PURPOSE:

A Round Robin Scheduler algorithm is designed especially for time sharing systems. It is similar to FCFS scheduling but preemption is added to switch between processes. Preemption: The act of interrupting a currently running task in order to give time to another task.

DESCRIPTION:

To implement Round robin scheduling we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a **timer to interrupt** after one time quantum and dispatches the process. A small unit of time called a time quantum or time slice is defined. A time quantum is generally from 10 to 100 milliseconds.

The process may have CPU burst of less than one time quantum. In this case the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. The average waiting time under Round Robin policy is how ever quite long.

Lab 12 Part II:

Threads scheduling

Write a program that checks the default scheduling policy of threads. Change the default policy to FIFO.

Functions to be used:

```
int pthread_attr_getschedpolicy(pthread_attr_t *attr, int policy);  
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int *policy);
```

Operating Systems LAB 13

Simulation of Banker's Algorithm for Deadlock Avoidance

Objectives:

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks.
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

Problem Statement: Implement Banker's algorithm for Deadlock Avoidance using C Language as per the algorithm given below.

A deadlock avoidance algorithm requires each process to make known in advance the maximum number of resources of each type that it may need. Also known is the maximum number of resources of each type available. Using both the above a priori knowledge, deadlock avoidance algorithm ensures that a circular wait condition never occurs.

Safe State:

A system is said to be in a safe state if it can allocate resources upto the maximum available and is not in a state of deadlock.

A safe sequence of processes always ensures a safe state. A sequence of processes $\langle P_1, P_2, \dots, P_n \rangle$ is safe for the current allocation of resources to processes if resource requests from each P_i can be satisfied from the currently available resources and the resources held by all P_j . If the state is safe then P_i requesting for resources can wait till P_j have completed. If such a safe sequence does not exist, then the system is in an unsafe state.

The resource allocation graph algorithm is not applicable where resources have multiple instances. In such a case Banker's algorithm is used. A new process entering the system must make known a priori the maximum instances of each resource that it needs subject to the maximum available for each type. As execution proceeds and requests are made, the system checks to see if the allocation of the requested resources ensures a safe state. If so only are the allocations made, else processes must wait for resources.

The following are the data structures maintained to implement the Banker's algorithm:

1. n : Number of processes in the system.
 2. m : Number of resource types in the system.
 3. Available: is a vector of length m . Each entry in this vector gives maximum instances of a resource type that are available at the instant.
- Available[j] = k means to say there are k instances of the j th resource type R_j .

4. Max: is a demand vector of size $n \times m$. It defines the maximum needs of each resource by the process. $\text{Max}[i][j] = k$ says the i th process P_i can request for atmost k instances of the j th resource type R_j .

5. Allocation: is a $n \times m$ vector which at any instant defines the number of resources of each type currently allocated to each of the m processes.

If $\text{Allocation}[i][j] = k$ then i th process P_i is currently holding k instances of the j th resource type R_j .

6. Need: is also a $n \times m$ vector which gives the remaining needs of the processes. $\text{Need}[i][j] = k$ means the i th process P_i still needs k more instances of the j th resource type R_j . Thus $\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$.

Safety Algorithm:

Using the above defined data structures, the Banker's algorithm to find out if a system is in a safe state or not is described below:

1. Define a vector Work of length m and a vector Finish of length n .
2. Initialize $\text{Work} = \text{Available}$ and $\text{Finish}[i] = \text{false}$ for $i = 1, 2, \dots, n$.
3. Find an i such that
 - a. $\text{Finish}[i] = \text{false}$ and
 - b. $\text{Need}_i \leq \text{Work}$ (Need_i represents the i th row of the vector Need).If such an i does not exist, go to step 5.
4. $\text{Work} = \text{Work} + \text{Allocation}_i$
Go to step 3.
5. If $\text{finish}[i] = \text{true}$ for all i , then the system is in a safe state.

Resource-Request Algorithm:

Let Request_i be the vector representing the requests from a process P_i .

$\text{Request}_i[j] = k$ shows that process P_i wants k instances of the resource type R_j . The following is the algorithm to find out if a request by a process can immediately be granted:

1. If $\text{Request}_i \leq \text{Need}_i$, go to step 2.
else Error "request of P_i exceeds Max_i ".
2. If $\text{Request}_i \leq \text{Available}_i$, go to step 3.
else P_i must wait for resources to be released.
3. An assumed allocation is made as follows:
 $\text{Available} = \text{Available} - \text{Request}_i$
 $\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$
 $\text{Need}_i = \text{Need}_i - \text{Request}_i$

If the resulting state is safe, then process P_i is allocated the resources and the above changes are made permanent. If the new state is unsafe, then P_i must wait and the old status of the data structures is restored.