**Practical 1**

**Aim**: Implement copy command using Open, Create, Read, Write, Access and Close system call.

**Hardware Requirement**:A Pentium-class processor with minimum 1 GB of RAM

**Software Requirement:** Unix supported Operating System

**Theory**:

**I/O System calls**

Basically, there are total 5 types of I/O system calls:

1. **Create:**Used to Create a new empty file.

**Syntax in C language:**

int create(char \*filename, mode\_t mode)

**Parameter:**

* + **filename:** name of the file which you want to create
  + **mode:** indicates permissions of new file.

**Returns:**

* + return first unused file descriptor (generally 3 when first create use in process because 0, 1, 2 fd are reserved)
  + return -1 when error

**How it works in OS**

* + Create new empty file on disk
  + Create file table entry
  + Set first unused file descriptor to point to file table entry
  + Return file descriptor used, -1 upon failure

1. **open**: Used to Open the file for reading, writing or both.

**Syntax in C language**

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

int open (const char\* Path, int flags [, int mode ]);

**Parameters:**

* + **Path:** path to file which you want to use
    - use absolute path begin with “/”, when you are not work in same directory of file.
    - Use relative path which is only file name with extension, when you are work in same directory of file.
  + **flags:** How you like to use
    - **O\_RDONLY**: read only, **O\_WRONLY**: write only, **O\_RDWR**: read and write, **O\_CREAT**: create file if it doesn’t exist, **O\_EXCL**: prevent creation if it already exists

**How it works in OS:**

* + Find existing file on disk
  + Create file table entry
  + Set first unused file descriptor to point to file table entry
  + Return file descriptor used, -1 upon failure

1. **close:**Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.

**Syntax in C language**

#include <fcntl.h>

int close(int fd);

**Parameter:**

* + **fd:** file descriptor

**Return:**

* + **0** on success.
  + **-1** on error.

**How it works in the OS:**

* + Destroy file table entry referenced by element fd of file descriptor table  
    – As long as no other process is pointing to it!
  + Set element fd of file descriptor table to **NULL**

1. **read:**From the file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by buf. A successful read() updates the access time for the file.

**Syntax in C language**

size\_t read (int fd, void\* buf, size\_t cnt);

**Parameters:**

* + **fd:** file descriptor
  + **buf:** buffer to read data from
  + **cnt:** length of buffer

**Returns: How many bytes were actually read**

* + return Number of bytes read on success
  + return 0 on reaching end of file
  + return -1 on error
  + return -1 on signal interrupt

**Important points:**

* + **buf** needs to point to a valid memory location with length not smaller than the specified size because of overflow.
  + **fd** should be a valid file descriptor returned from open() to perform read operation because if fd is NULL then read should generate error.
  + **cnt** is the requested number of bytes read, while the return value is the actual number of bytes read. Also, some times read system call should read less bytes than cnt.

1. **write:**Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT\_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

#include <fcntl.h>

size\_t write (int fd, void\* buf, size\_t cnt);

**Parameters:**

* + **fd:** file descriptor
  + **buf:** buffer to write data to
  + **cnt:** length of buffer

**Returns: How many bytes were actually Written**

* + return Number of bytes written on success
  + return 0 on reaching end of file
  + return -1 on error
  + return -1 on signal interrupt

**Important points:**

* + The file needs to be opened for write operations
  + **buf**needs to be at least as long as specified by cnt because if buf size less than the cnt then buf will lead to the overflow condition.
  + **cnt** is the requested number of bytes to write, while the return value is the actual number of bytes written. This happens when **fd** have a less number of bytes to write than cnt.
  + If write() is interrupted by a signal, the effect is one of the following:  
    -If write() has not written any data yet, it returns -1 and sets errno to EINTR.  
    -If write() has successfully written some data, it returns the number of bytes it wrote before it was interrupted.

**Code**:

#include<stdio.h>

#include<string.h>

#include<unistd.h>

#include<fcntl.h>

int main(int argc, char \*argv[])

{

int i, fd[2],sz;

char \*c = (char \*) calloc (10, sizeof(char));

fd[0] = open(argv[1], O\_RDONLY, 0);

fd[1] = open(argv[2], O\_WRONLY, 0);

while((i = read(fd[0],c,10))>0)

{

write(fd[1], c, i);

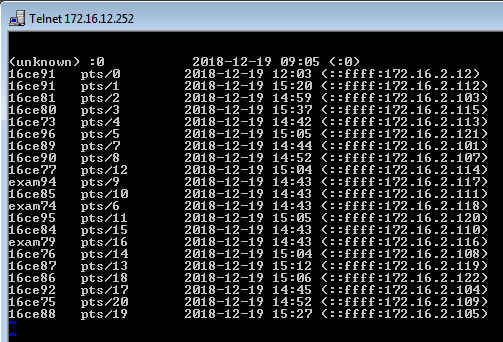
}

close(fd[0]);

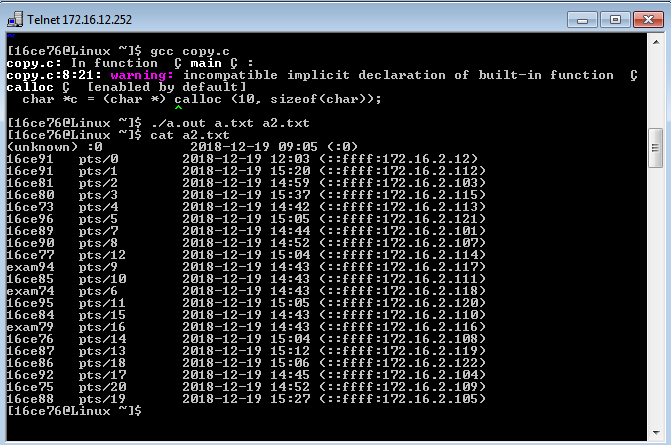
close(fd[1]);

}

a.txt : **The text file with content**



**Output**: After running copy command the content from file a.txt is copied into a2.txt



**Conclusion** :

In a file system from the copy command, we can keep the original file and make a duplicate of it. From this practical, we have studied how copy command works in internal by the operating system.

**Practical 2**

**Aim**: Write a program that creates a file with a hole in it using lseek().

**Hardware Requirement**:A Pentium-class processor with minimum 1 GB of RAM

**Software Requirement:** Unix supported Operating System

**Theory**:

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

**SEEK\_SET**

The offset is set to *offset* bytes.

**SEEK\_CUR**

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

**SEEK\_END**

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

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

Since version 3.1, Linux supports the following additional values for*whence*:

**SEEK\_DATA**

Adjust the file offset to the next location in the file greater than or equal to *offset* containing data. If *offset* points to data, then the file offset is set to *offset*.

**SEEK\_HOLE**

Adjust the file offset to the next hole in the file greater than or equal to *offset*. If *offset* points into the middle of a hole, then the file offset is set to *offset*. If there is no hole past *offset*, then the file offset is adjusted to the end of the file (i.e., there is an implicit hole at the end of any file).

In both of the above cases, **lseek**() fails if *offset* points past the end of the file.

**RETURN VALUE**

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

**Code:**

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <stdlib.h>

int main(void)

{

int fd;

int ret;

fd = open("hole.txt",O\_WRONLY|O\_CREAT,1024);

write(fd,"hello",5);

ret = lseek(fd,10,SEEK\_CUR);

write(fd,"world",5);

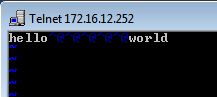
close(fd);

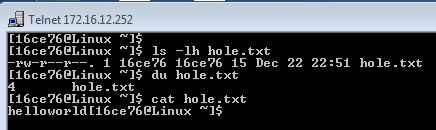
return 0;

}

**Output:**

Hole.txt :





**Conclusion**:

From this practical, we have learned that lseek command is used to change the location of the read/write pointer of a file descriptor.

**Practical 3**

**Aim**: You need to create user defined function to create processes and to join those processes using fork, wait system call. Write a program to demonstrate file sharing among child and parent.

**Hardware Requirement**:A Pentium-class processor with minimum 1 GB of RAM

**Software Requirement:** Unix supported Operating System

**Theory**:

**Fork System Call:**

Fork system call use for creates a new process, which is called **child process**, which runs concurrently with process (which process called system call fork) and this process is called **parent process**. After a new child process created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

**Negative Value**: creation of a child process was unsuccessful.

**Zero**: Returned to the newly created child process.

**Positive value**: Returned to parent or caller. The value contains process ID of newly created child process.

**Important:** Parent process and child process are running the same program, but it does not mean they are identical. OS allocate different data and state for these two processes and also control the flow of these processes can be different. See next example

**Wait System Call:**

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution after wait system call instruction.  
Child process may terminate due to any of these:

* It calls exit();
* It returns (an int) from main
* It receives a signal (from the OS or another process) whose default action is to terminate.

If any process has more than one child processes, then after calling wait(), parent process has to be in wait state if no child terminates.

If only one child process is terminated, then return a wait() returns process ID of the terminated child process.

If more than one child processes are terminated than wait() reap any arbitrarily child and return a process ID of that child process.

When wait() returns they also define exit status (which tells our, a process why terminated) via pointer, If status are not NULL.

If any process has no child process then wait() returns immediately “-1”

Code:

Header File

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

int process\_create(int nop){

    int i;

    for(i=1;i<=nop;i++){

        if(fork()==0)

        {

          return i;

        }

    }

   return 0;

}

void process\_join(int nop,int id)

{

  int i;

if(id==0)

{

  for(i=1;i<nop;i++){

    wait(0);

    }

}

else{

        exit(0);

}

}

Main file

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include "header.h"

int main()

{

int id;

printf("Hi\n");

  id= process\_create(5);

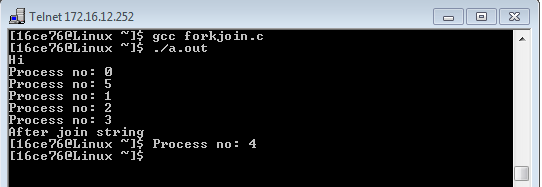
    printf("Process no: %d\n",id);

     process\_join(5,id);

printf("After join string\n");

}

Output:



Conclusion:

From this practical we have learned how fork system call creates child process from the parent process and how a call to wait() blocks the calling process until one of its child processes exits or a signal is received

**Practical 4**

**Aim**: Write a program to implement Zombie process and Orphan process

**Hardware Requirement**:A Pentium-class processor with minimum 1 GB of RAM

**Software Requirement:** Unix supported Operating System

**Theory**:

**Orphan Process:**

An orphan process is a computer **process whose parent process has finished or terminated**, though it remains running itself.

In a Unix-like operating system any orphaned process will be immediately adopted by the special init system process. This operation is called re-parenting and occurs automatically. Even though technically the process has the init process as its parent, it is still called an orphan process since the process that originally created it no longer exists.

A process can be orphaned unintentionally, such as when the parent process terminates or crashes. The process group mechanism in most Unix-like operation systems can be used to help protect against accidental orphaning, where in coordination with the user’s shell will try to terminate all the child processes with the SIGHUP process signal, rather than letting them continue to run as orphans.

A process may also be intentionally orphaned so that it becomes detached from the user’s session and left running in the background; usually to allow a long-running job to complete without further user attention, or to start an indefinitely running service. Under Unix, the latter kinds of processes are typically called daemon processes. The Unix nohup command is one means to accomplish this.

**Daemon Process:**

In Unix and other multitasking computer operating systems, a **daemon is a computer program that runs as a background process**, rather than being under the direct control of an interactive user. Typically daemon names end with the letter d: for example,syslogd is the daemon that implements the system logging facility and sshd is a daemon that services incoming SSH connections.

In a Unix environment, the parent process of a daemon is often, but not always, the init process. A daemon is usually created by a process forking a child process and then immediately exiting, thus causing init to adopt the child process. In addition, a daemon or the operating system typically must perform other operations, such as dissociating the process from any controlling terminal (tty). Such procedures are often implemented in various convenience routines such as daemon(3) in Unix.

**Zombie Process:**

On Unix and Unix-like computer operating systems, **a zombie process or defunct process is a process that has completed execution but still has an entry in the process table**. This entry is still needed to allow the parent process to read its child’s exit status. The term zombie process derives from the common definition of zombie — an undead person. In the term’s metaphor, the child process has “died” but has not yet been “reaped”. Also, unlike normal processes, the kill command has no effect on a zombie process.

When a process ends, all of the memory and resources associated with it are deallocated so they can be used by other processes. However, the process’s entry in the process table remains. The parent can read the child’s exit status by executing the wait system call, whereupon the zombie is removed. The wait call may be executed in sequential code, but it is commonly executed in a handler for the SIGCHLD signal, which the parent receives whenever a child has died.

After the zombie is removed, its process identifier (PID) and entry in the process table can then be reused. However, if a parent fails to call wait, the zombie will be left in the process table. In some situations this may be desirable, for example if the parent creates another child process it ensures that it will not be allocated the same PID. On modern UNIX-like systems (that comply with SUSv3 specification in this respect), the following special case applies: if the parent explicitly ignores SIGCHLD by setting its handler to SIG\_IGN (rather than simply ignoring the signal by default) or has the SA\_NOCLDWAIT flag set, all child exit status information will be discarded and no zombie processes will be left

A **zombie process is not the same as an orphan process**. An orphan process is a process that is still executing, but whose parent has died. They do not become zombie processes; instead, they are adopted by init (process ID 1), which waits on its children.

Code:

Orphan Process

#include<stdio.h>

#include<unistd.h>

int main()

{

printf("\n Parent\_id: %d",getpid());

if(fork()==0)

{ printf("\nChild\_id: %d",getpid());

printf("\nParent id of the Child process: %d",getppid());

sleep(5);

printf("\n Child\_id: %d",getpid());

printf("\n Child's dead Parent's id : %d",getppid());

}

else

{

sleep(2);

exit(0);

}

}

**Zombie Process:**

#include<stdio.h>

#include<unistd.h>

int main()

{

printf("\n Parent id: %d",getpid());

if(fork()==0)

{ printf("\nCHild ID: %d",getpid());

printf("\n CHILD'S PARENT ID: %d",getppid());

printf("\n Child Destroyed");

exit(0);

}

else

{

printf("Child Borned");

sleep(50);

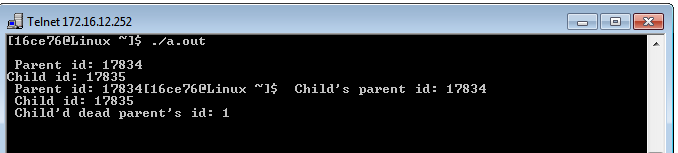
printf("This is the zombie process");

exit(0);

}

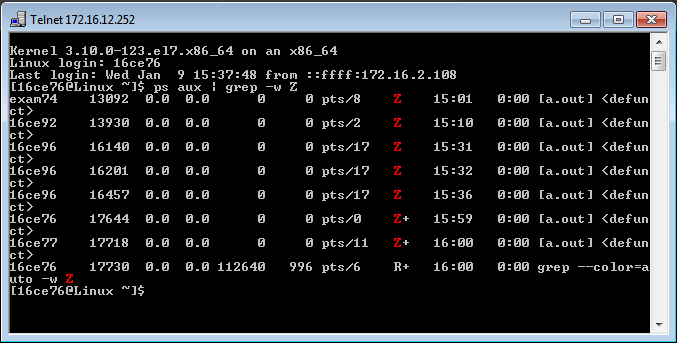
}

Orphan Process:



Zombie Process:





Conclusion:

From this practical, we have learned that, a  process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process and a process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

**Practical 5**

**Aim**: Implement below system calls:

1. ls (b) grep (c) head

**Hardware Requirement**:A Pentium-class processor with minimum 1 GB of RAM

**Software Requirement:** Unix supported Operating System

**Theory**:

1. **ls :**

The ls command lists the contents of, and optional information about, directories and files. With no options, ls lists the files contained in the current directory, sorting them alphabetically.

* What is the ls command?

The ls command is a command-line utility for listing the contents of a directory or directories given to it via standard input. It writes results to standard output. The ls command supports showing a variety of information about files, sorting on a range of options and recursive listing.

* How to show the contents of a directory

To show the contents of a directory pass the directory name to the ls command. This will list the contents of the directory in alphabetical order. If your terminal supports colours you may see that file and directory listings are a different colour.

Example : ls /home/george

bin code dotfiles Downloads go irc logs src

* How to show hidden files and folders

To show hidden files and folders pass the -a option to ls this will

Example: ls -a /home/george

. .goobook .tmux.conf

.. .goobook\_auth.json .urlview

.asoundrc .inputrc .vim

.asoundrc.asoundconf .install.sh .viminfo

.asoundrc.asoundconf.bak .irbrc .viminfo.tmp

...

1. **grep:**

* What is the grep command in UNIX?

The grep command in UNIX is a command line utility for printing lines that match a pattern. It can be used to find text in a file and search a directory structure of files recursively. It also supports showing the context of a match by showing lines before and after the result and has support for regular expressions in pattern matching.

* How to find text in a file

To find text in a file pass the string you are looking for to grep followed by the name of the file or files.

Example: grep 'computer' /usr/share/dict/words

computer

The grep tool will print occurrences that it finds to standard output.

* How to list line numbers for matches

To list line numbers and file names pass the -n option to grep. This prints matches to standard output along with the line number it was found on.

Example: grep 'computer' -n /usr/share/dict/words

40565

This can be useful if you are looking to edit a file and want to launch vim and go straight to the line.

vim +40565 /usr/share/dict/words

* How to count the number of matches

To count the number of matches use the -c option. This outputs a number count to standard output.

Example: grep -c 'comput\*' /usr/share/dict/words

50

* How to ignore case when searching

To ignore case when searching use the -i option. By default grep will respect case.

Example:

grep 'COMPUTER' /usr/share/dict/words

# no match

grep -i 'COMPUTER' /usr/share/dict/words

computer

1. **head**:

* What is the head command?

The head command is a command-line utility for outputting the first part of files given to it via standard input. It writes results to standard output. By default head returns the first ten lines of each file that it is given.

* How to view the first ten lines of a file

To view the first ten lines of a file pass the name of a file to the head command. The first ten lines of the file will be printed to standard output.

Example: head /usr/share/dict/words

A

a

AA

AAA

Aachen

aah

Aaliyah

Aaliyah's

aardvark

aardvark's

* How to limit the number of lines to show

To set the number of lines to show with head pass the -n option followed by the number of lines to show.

Example:

head -n 1 /usr/share/dict/words

A

* How to limit the number of bytes to show

To limit the number of bytes shown with head pass the -c option. Instead of limiting by number of lines this will limit by the number of bytes passed to the -c option. In the following example the output is limited to 16 bytes.

Example:

head -c 16 /usr/share/dict/words

A

a

AA

AAA

Aache%

Code:

1. ls:

#include <dirent.h>

#include <stdio.h>

#include <stdlib.h>

void main()

{

struct dirent \*\*namelist;

int n;

n = scandir(".", &namelist, NULL, alphasort);

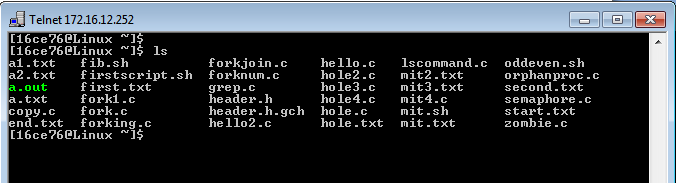
while (n--)

{

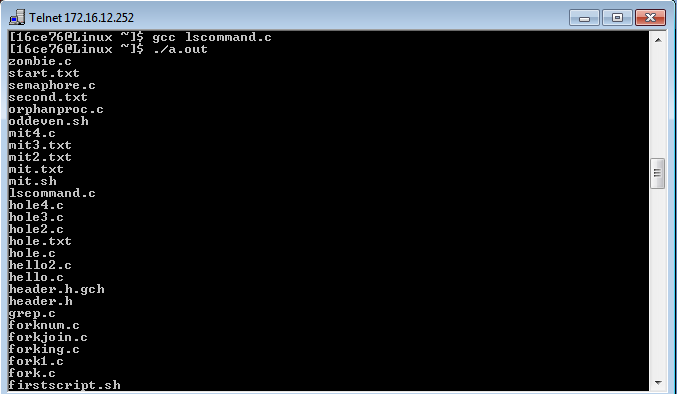
printf("%s\n", namelist[n]->d\_name);

}

}

**Output:** 

Normal ls command



ls command execution from program

(b) grep

#include<stdio.h>

#include<string.h>

#include<unistd.h>

int main(int argc, char \*argv[])

{

FILE \*fp;

char sentence[100];

fp = fopen(argv[2],"r");

while(fscanf(fp , "%[^\n]\n" , sentence)!=EOF)

{

if(strstr(sentence , argv[1]) !=NULL)

printf("%s\n" , sentence);

else

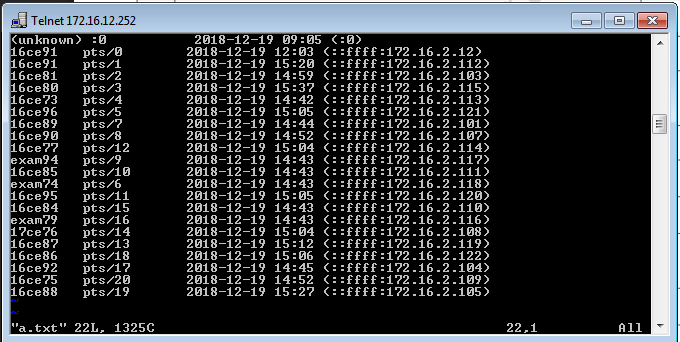
continue;

}

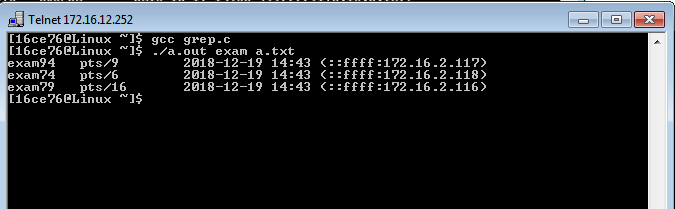
fclose(fp);

}

**Output:**



File a.txt

Searching" exam" in a.txt

**c) head**

**Code:**

#include<stdio.h>

#include<string.h>

#include<unistd.h>

int main(int argc, char \*argv[])

{

FILE \*fp;

fp = fopen(argv[1],"r");

char line[256];

int l=10;

if (argc == 3)

{

l = atoi(argv[2]);

}

int i = 0;

while (fgets(line,sizeof line, fp)!= NULL)

{

if(i<l)

{

fscanf(fp,"%[^\n]", line);

printf("%s\n",line);

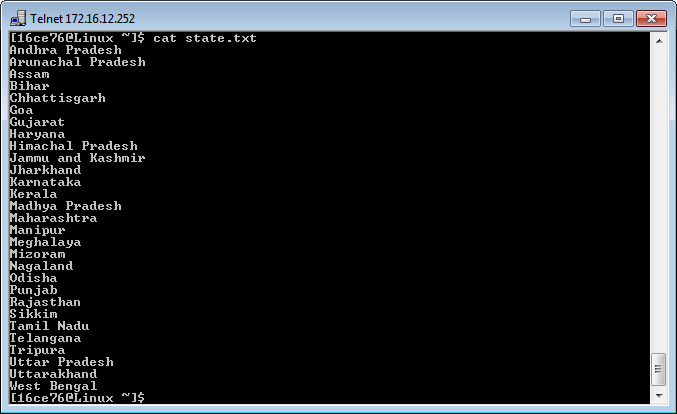
i++;

}

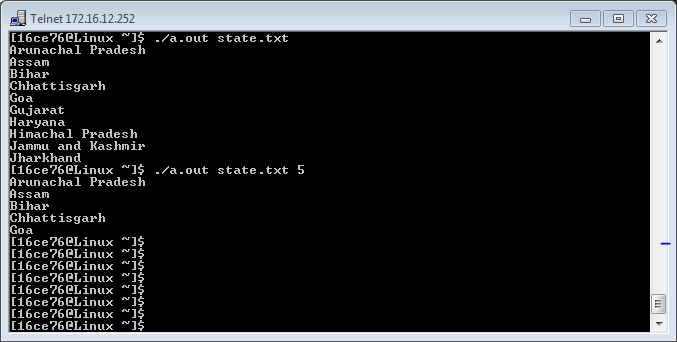
}

fclose(fp);

}

**Output:** 

File "State.txt"



head command implementation for default 10 lines and with user given lines

**Conclusion:**

From this practical we have learned more 3 basic commands ls, grep and head which are most useful commands in linux. We have implemented this commands with C logic and with command line arguments.

**Practical 6**

**Aim**: Write a program to perform input /output redirection from/to file using dup().

**Hardware Requirement**:A Pentium-class processor with minimum 1 GB of RAM

**Software Requirement:** Unix supported Operating System

**Theory**:

**What is a dup() system call**

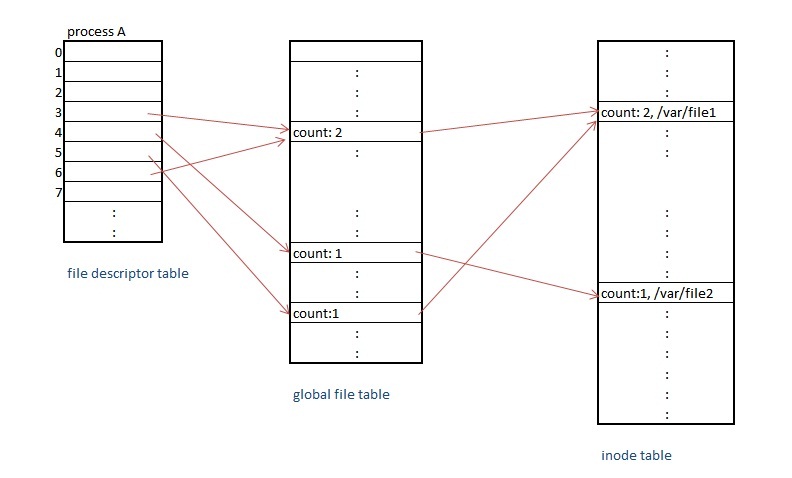
dup() system call in unix systems copies a file descriptor into the first free slot in the private file descriptor table and then returns the new file descriptor to the user. It works for all the file types. The syntax is :

newfd = dup(fd);

Here fd is the file descriptor being duped and newfd is returned to the user.

There are basically three different data structures that helps in manipulation of file system. These are - the inode table, private user file descriptor table and the global file table. Before moving forward to the description of dup() command, I urge you to please follow this article on Internal Data Structure for file handling in Unix kernel.

dup() system call doesnt create a separate entry in the global file table like the open() system call, instead it just increments the count field of the entry pointed to by the given input file descriptor in the global file table. Consider an example where fd 0, 1 and 2 are by default engaged to the standard input/output and error. Then if the user opens a file "/var/file1" (fd - 3), then he opens file "/var/file2" (fd - 4) and again he opened "/var/file1" (fd - 5). And now, if he does a dup(3), kernel would follow the pointer from the user file descriptive table for the fd entry '3', and increments the count value in the global file table. Then, it searches for the next avaialable free entry in file descriptor table and returns that value to the user (6 in this case).



**Input / output redirection using dup() system call**

dup() system call finds use in implementing input/output redirection or piping the output on unix shell. Suppose, we wish to redirect the output of 'ls' command to a file, we use the following command on shell to do our job:

root> ls /var/\* > tempfile

File descriptor 1 is bound to the standard output stream. The 'ls /var/\*' command is supposed to output the data on this output stream i.e. 1. But, using '>' operator we are able to redirect this output to file 'tempfile'. What happens when the process that is executing the shell here is that it parses the command and when it finds '>' operator, it will first find the file descriptor of the rhs operand - 'tempfile' OR create the new fd if file doesnt exist already. Once, it finds this fd, it will close the stdout file descriptor and call a dup() on the given fd for this 'tempfile'.

Thats it, from this step onwards, the output will be redirected to the file 'tempfile'. We can also do an additional step of closing the file descriptor to preserve the number of descriptors.

/\*redirection of I/O\*/

{

fd = creat('tempfile', flags);

close(stdout); //stdout => 1

dup(fd);

close(fd);

/\* stdout is now redirected \*/

}

The same logic is applied when we apply "pipe" operations on the shell. Thus, although dup() is not an elegant command but yet it is a powerful building block for several higher level commands.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <string.h>

int main(int argc, char \*argv[])

{

int i,j;

char buf[512];

int fd1 = open("read.txt",O\_RDONLY);

int fd2 = open("write.txt",O\_WRONLY);

int r = dup(fd1);

int w = dup(fd2);

close(fd1);

close(fd2);

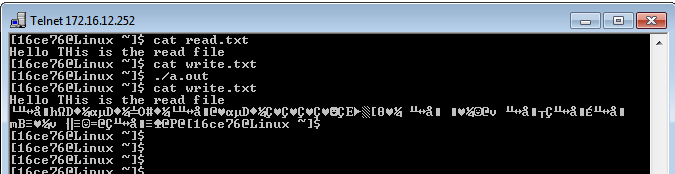
read(r, buf, sizeof(buf));

write(w, buf, sizeof(buf));

return 0;

}

**Output:**



**Conclusion:**

From this practical we have learned dup system call copies a file descriptor into the first free slot in the private file descriptor table and then returns the new file descriptor to the user. We have implemented this logic in C program.

**Practical 7**

**Aim**: Write a program to perform addition of 1 to 100. Inter Process Communication using Shared Memory and Pipe. You need to us shmget and shmat system call.

**Hardware Requirement**:A Pentium-class processor with minimum 1 GB of RAM

**Software Requirement:** Unix supported Operating System

**Theory**:

**IPC through Pipe**

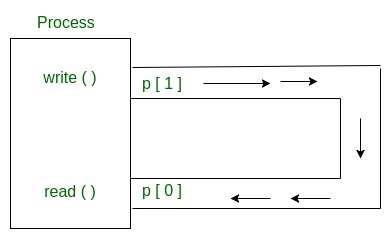
Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. In UNIX Operating System, Pipes are useful for communication between related processes(inter-process communication).

Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a “virtual file”.

The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this “virtual file” or pipe and another related process can read from it.

If a process tries to read before something is written to the pipe, the process is suspended until something is written.

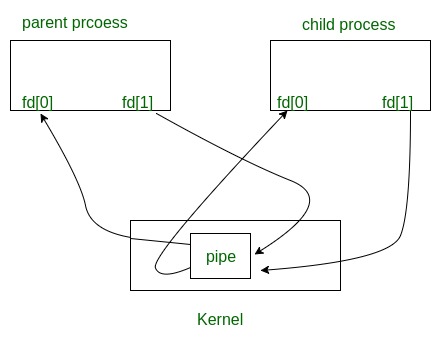
The pipe system call finds the first two available positions in the process’s open file table and allocates them for the read and write ends of the pipe.



Pipes behave FIFO(First in First out), Pipe behave like a queue data structure. Size of read and write don’t have to match here. We can write 512 bytes at a time but we can read only 1 byte at a time in a pipe.

**Parent and child sharing a pipe:**

When we use fork in any process, file descriptors remain open across child process and also parent process. If we call fork after creating a pipe, then the parent and child can communicate via the pipe.



Code:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#define P1\_READ 0

#define P2\_WRITE 1

#define P2\_READ 2

#define P1\_WRITE 3

#define NUM\_PIPES 2

int main(int argc, char \*argv[])

{

int fd[2\*NUM\_PIPES];

int val = 0, len, i;

pid\_t pid;

for (i=0; i<NUM\_PIPES; ++i)

{

if (pipe(fd+(i\*2)) < 0)

{

perror("Failed to allocate pipes");

exit(EXIT\_FAILURE);

}

}

if ((pid = fork()) < 0)

{

perror("Failed to fork process");

return EXIT\_FAILURE;

}

if (pid == 0)

{

close(fd[P1\_READ]);

close(fd[P1\_WRITE]);

pid = getpid();

len = read(fd[P2\_READ], &val, sizeof(val));

if (len < 0)

{

perror("Child: Failed to read data from pipe");

exit(EXIT\_FAILURE);

}

else if (len == 0)

{

fprintf(stderr, "Child: Read EOF from pipe");

}

else

{

printf("Child(%d): Received %d\n", pid, val);

val \*= 2;

printf("Child(%d): Sending %d back\n", pid, val);

if (write(fd[P2\_WRITE], &val, sizeof(val)) < 0)

{

perror("Child: Failed to write response value");

exit(EXIT\_FAILURE);

}

}

close(fd[P2\_READ]);

close(fd[P2\_WRITE]);

return EXIT\_SUCCESS;

}

close(fd[P2\_READ]);

close(fd[P2\_WRITE]);

pid = getpid();

val = 42;

printf("Parent(%d): Sending %d to child\n", pid, val);

if (write(fd[P1\_WRITE], &val, sizeof(val)) != sizeof(val))

{

perror("Parent: Failed to send value to child ");

exit(EXIT\_FAILURE);

}

len = read(fd[P1\_READ], &val, sizeof(val));

if (len < 0)

{

perror("Parent: failed to read value from pipe");

exit(EXIT\_FAILURE);

}

else if (len == 0)

{

fprintf(stderr, "Parent(%d): Read EOF from pipe", pid);

}

else

{

printf("Parent(%d): Received %d\n", pid, val);

}

close(fd[P1\_READ]);

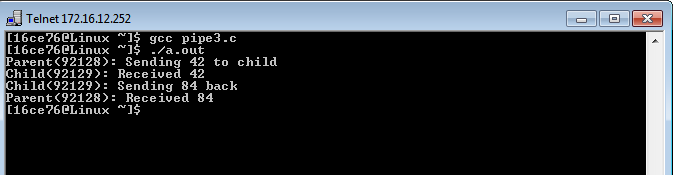
close(fd[P1\_WRITE]);

wait(NULL);

return EXIT\_SUCCESS;

}

Output:



**IPC through shared memory**

Inter Process Communication through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by anther process.

The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

* Server reads from the input file.
* The server writes this data in a message using either a pipe, fifo or message queue.
* The client reads the data from the IPC channel,again requiring the data to be copied from kernel’s IPC buffer to the client’s buffer.
* Finally the data is copied from the client’s buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

SYSTEM CALLS USED ARE:

**ftok():** is use to generate a unique key.

**shmget():** int shmget(key\_t,size\_tsize,intshmflg); upon successful completion, shmget() returns an identifier for the shared memory segment.

**shmat():** Before you can use a shared memory segment, you have to attach yourself

to it using shmat(). void \*shmat(int shmid ,void \*shmaddr ,int shmflg);

shmid is shared memory id. shmaddr specifies specific address to use but we should set

it to zero and OS will automatically choose the address.

**shmdt():** When you’re done with the shared memory segment, your program should

detach itself from it using shmdt(). int shmdt(void \*shmaddr);

**shmctl():** when you detach from shared memory,it is not destroyed. So, to destroy

shmctl() is used. shmctl(int shmid,IPC\_RMID,NULL);

**Code:**

#include <stdio.h>

#include <sys/types.h>

#include <sys/shm.h>

#include <unistd.h>

int main(int argc, char \*\*argv)

{

pid\_t child;

int shmid;

int\* shmptr;

shmid = shmget((key\_t) 1234, 3 \* sizeof(int), 0666 | IPC\_CREAT);

if (shmid == -1)

{

perror("shmget");

return -1;

}

shmptr = (int\*) shmat(shmid, NULL, 0);

if (shmptr == (void\*) -1)

{

perror("shmat");

return -1;

}

shmptr[2] = 0;

child = fork();

if (child == -1)

{

perror("fork");

return -1;

}

if (child > 0)

{

waitpid(child, NULL, 0);

int s = shmptr[0], e = shmptr[1], r = shmptr[2];

printf("Sum of numbers from %d to %d is %d.\n", s, e, r);

}

else if (child == 0)

{

int i;

scanf("%d %d", &shmptr[0], &shmptr[1]);

for (i = shmptr[0]; i <= shmptr[1]; i++)

shmptr[2] += i;

return 0;

}

if (shmdt(shmptr) == -1)

{

perror("shmdt");

return -1;

}

if (shmctl(shmid, IPC\_RMID, 0) == -1)

{

perror("shm remove");

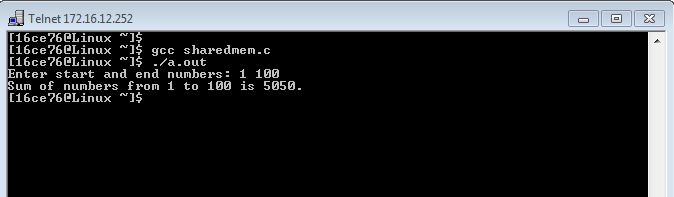
return -1;

}

return 0;

}

**Output:**



**Conclusion:**

From this practical we have learned pipe system call and shared memory to handle Inter process communication.

**Practical 8**

**Aim**: Implement below file system calls: bmap

**Hardware Requirement**:A Pentium-class processor with minimum 1 GB of RAM

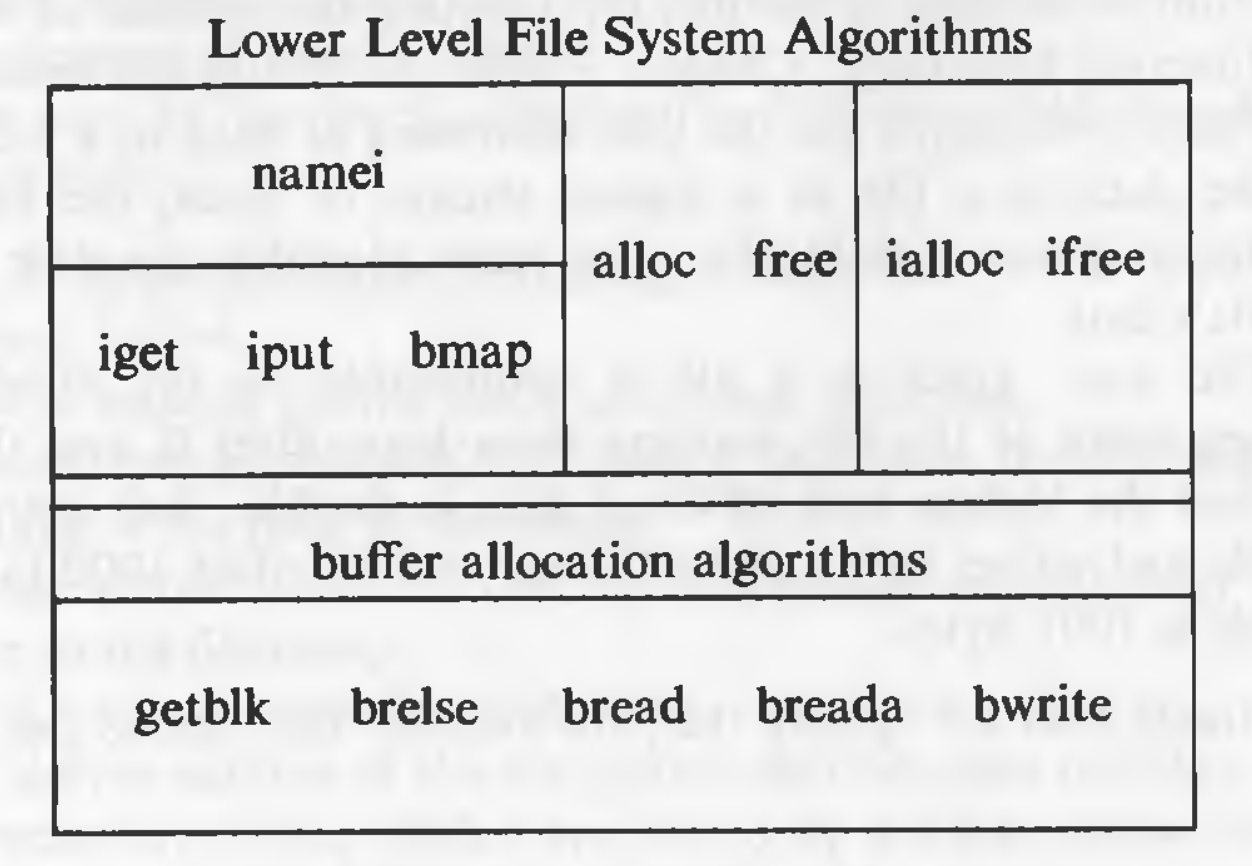
**Software Requirement:** Unix supported Operating System

**Theory**:

# Internal Representation of Files

Every file a UNIX system has a unique inode. Processes interact with files using well defined system calls. The users specify a file with a character string which is the file's path and then the system get the inode which is mapped to the file which corresponds to the path.

The algorithms described below are above the layer of buffer cache. Diagrammatically, it can be shown like this:

[](https://github.com/suvratapte/Maurice-Bach-Notes/blob/master/Diagrams/Screen_Shot_2017-06-08_at_9.29.01_PM.png)

## Inodes

Inodes exist in a static form on the disk. The kernel reads them into in-core inodes and modifies them.

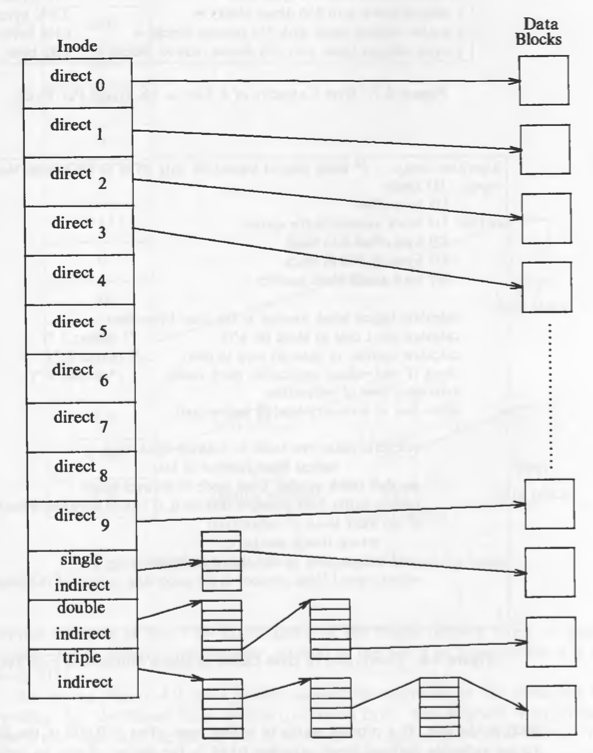
Disk inodes consists of the following fields:

* Owner information: ownership is divided into a user and a group of users. Root user has access to all the files.
* File type: it states whether a file is a normal file, a directory, a block or character special file, or a device file.
* File access permissions: there are 3 types of access permissions: owner, group and others. There are separate permissions for reading, writing and executing. Since execute permission is not applicable to a directory, execute permission for a directory gives the right to search inside the directory.
* Access times: the times at which the file was last accessed and last modified, and the time at which the inodes was last modified
* Number of links: number of places from which the file is being referred.
* Array of disk blocks: even if the users get a logically sequential representation of data in files, the actual data is scattered across the disk. This array keeps the addresses of the disk blocks on which the data is scattered.
* File size: the addressing of the file begins from location 0 from relative to the starting location and the size of the file is the maximum offset of the file + 1. For example, if a user creates a file and writes a byte at offset 999, the size of the file is 1000.

## Structure of a Regular File

In UNIX, the data in files is not stored sequentially on disk. If it was to be stored sequentially, the file size would not be flexible without large fragmentation. In case of sequential storage, the inode would only need to store the starting address and size. Instead, the inode stores the disk block numbers on which the data is present. But for such strategy, if a file had data across 1000 blocks, the inode would need to store the numbers of 1000 blocks and the size of the inode would differ according to the size of the file.

To be able to have constant size and yet allow large files, indirect addressing is used. The inodes have array of size 13 which for storing the block numbers, although, the number of elements in array is independent of the storage strategy. The first 10 members of the array are "direct addresses", meaning that they store the block numbers of actual data. The 11th member is "single indirect", it stores the block number of the block which has "direct addresses". The 12th member is "double indirect", it stores block number of a "single indirect" block. And the 13th member is "triple indirect", it stores block number of a "double indirect" block. This strategy can be extended to "quadruple" or "quintuple" indirect addressing.

[](https://github.com/suvratapte/Maurice-Bach-Notes/blob/master/Diagrams/Screen_Shot_2017-06-09_at_4.07.02_PM.png)

If a logical block on the file system holds 1K bytes and that a block number is addressable by a 32 bit integer, then a block can hold up to 256 block numbers. The maximum file size with 13 member data array is:

10 direct blocks with 1K bytes each = 10K bytes

1 indirect block with 256 direct blocks = 256K bytes

1 double indirect block with 256 indirect blocks = 64M bytes

1 triple indirect block with 256 double indirect blocks = 16G bytes

Code:

#include<stdio.h>

void main()

{

long int offset;

printf("Enter byteoffset : ");

scanf("%d",&offset);

int b\_offset;

int single=-1,doubles=-1,triples=-1;

//Value -1 indicates that there is no redirect of that type

long int block=offset/1024;

if(block < 266)

{

single=block;

b\_offset=offset%1024;

goto SKP;

}

else if(block > 266 && block < 65802)

{

single=block/256;

doubles=block%256-10;

b\_offset=offset%1024;

goto SKP;

}

else if(block > 65802 && block<10000000)

{

single=block/(256\*256);

long int temp=block%(256\*256);

doubles=temp/256;

triples=temp%256;

b\_offset=offset%1024;

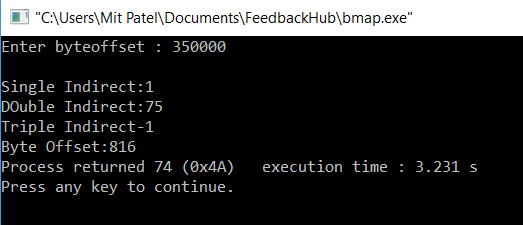
}

SKP:

printf("\nSingle Indirect:%d \nDOuble Indirect:%d \nTriple Indirect%d \nByte Offset:%d",single,doubles,triples,b\_offset);

}

Output:



Conclusion:

Bmap algorithm is used for Unix operating system and it can hold up to 4 GB of file size.