**OPERATING SYSTEM**

**PROGRAM 1**

Aim:

**Q)Familiarity with Lab environment and operating system, User and System level Operating System Commands**

**Description**

1. **ls:**  list directory contents

List information about the files (the current directory by default).

Sort entries alphabetically.

**Syntax:**

ls [OPTION] … [FILE]…

1. **pwd:** print name of the current working directory.

**Syntax:**

pwd [OPTION]…

1. **mkdir:**  create the directory(ies) if they don’t already exist.

**Syntax:**

mkdir [OPTION]… DIRECTORY

1. **cd:** change the working directory

**Syntax:**

cd [-L|-P] [director]

1. **cp:** copy files and directories

copy source to destination or multiple source(s) to Directory

**Syntax:**

cp [OPTION] … [-T] SOURCE DEST

cp [OPTION] … SOURCE … DIRECTORY

1. **mv:** Rename Source to Destination or more Source(s) to Directory.

**Syntax:**

mv [OPTION] … [-T] SOURCE DEST

mv [OPTON] … SOURCE … DIRECTORY

mv [OPTION] … -t DIRECTORY SOURCE

1. **rm:** remove files or directories.

**Syntax:**

rm [OPTION] … [FILE]

1. **clear:** Clear Terminal
2. **cat:** Display file contents on terminal

**Syntax:**

cat [OPTION] .. [FILE]

1. **whoami:** Reveals current username associated with the active user session.

**Syntax:** Whoami [OPTION] ….

**PROGRAM-2:**

**Aim:**

Program to demonstrate usage of linux system calls on: File management: open (), read (), write (), close ().

**Description:**

Linux system calls for file management provide the necessary interface for creating, reading, writing, and manipulating files within the operating system. These system calls enable programs to perform operations such as opening files, closing files, reading data from files, writing data to files, and manipulating file metadata such as permissions and ownership. Examples of file management system calls in Linux include open(), close(), read(), write(), mkdir(), rmdir(), rename(), and chmod().

1. **open():** 
   * **Purpose:** Opens or creates a file.
   * **Syntax:** int open(const char \*pathname, int flags, mode\_t mode); • **Example:** fd = open("example.txt", O\_RDWR | O\_CREAT, 0644); if (fd == -1) {

perror("open"); exit(EXIT\_FAILURE);

}

1. **read():** 
   * **Purpose:** Reads data from an open file descriptor into a buffer.
   * **Syntax:** ssize\_t read(int fd, void \*buf, size\_t count);
   * **Example:**

ssize\_t bytes\_read = read(fd, buf, sizeof(buf)); if (bytes\_read == -1) { perror("read"); exit(EXIT\_FAILURE);

}

1. **write():** 
   * **Purpose:** Writes data from a buffer to an open file descriptor.
   * **Syntax:** ssize\_t write(int fd, const void \*buf, size\_t count); • **Example:**

ssize\_t bytes\_written = write(fd, msg, strlen(msg)); if (bytes\_written == -1) {

perror("write"); exit(EXIT\_FAILURE);

}

1. **close():** 
   * **Purpose:** Closes an open file descriptor.
   * **Syntax:** int close(int fd); • **Example:**

if (close(fd) == -1) { perror("close"); exit(EXIT\_FAILURE);

}

**PROGRAM 2**

**Q) Program to demonstrate usage of linux system calls on:**

**File management: open (), read (), write (), close ()**

**PROGRAM:**

#include<stdio.h>

#include<fcntl.h>

#include<stdlib.h>

#include<string.h>

int main()

{

int fp=open("f1.txt",O\_RDWR|O\_CREAT,0666);

if(fp==-1)

{

printf("file cannot be opened");

exit(1);

}

printf("file has been opened with descriptor %d\n",fp);

char \*str="please write this line to file";

int rval=write(fp,str,strlen(str));

if(rval==-1)

{

printf("unnable to write");

exit(1);

}

printf("%d characters written to file",rval);

close(fp);

**OUTPUT:**

file.PNG

**PROGRAM-3:**

**Aim:**

Program to Display process information using process related system Calls: fork(), getpid(), getppid().

**Description:**

Displaying process information using process-related system calls involves retrieving details about currently running processes in a system. This typically includes information such as process IDs (PIDs), parent process IDs (PPIDs), process states, CPU and memory usage, execution priority, and other relevant data. System calls like getpid(), getppid(), getpriority(), getrusage(), and stat() provide access to this information, enabling monitoring and management of processes within an operating system.

1. **fork():** 
   * **Purpose:** Creates a new process by duplicating the calling process.
   * **Syntax:** pid\_t fork(void); • **Example:**

#include <stdio.h> #include <unistd.h> int main() {

fork();

printf("PID: %d\n", getpid()); return 0;

}

1. **getpid():** 
   * **Purpose:** Returns the process ID (PID) of the calling process. • **Syntax:** pid\_t getpid(void);
   * **Example:**

#include <stdio.h> #include <unistd.h> int main() {

printf("Process ID: %d\n", getpid());

return 0;

}

1. **getppid():** 
   * **Purpose:** Returns the parent process ID (PPID) of the calling process.
   * **Syntax:** pid\_t getppid(void); • **Example:**

#include <stdio.h> #include <unistd.h> int main() {

printf("PPID: %d\n", getppid());

return 0;

}

**PROGRAM 3**

**Q)Program to Display process information using process related system calls fork(), getpid(), getppid()**

**PROGRAM:**

#include<stdio.h>

int main()

{

printf("\nThis is parent process\n");

int pid=fork();

if(pid>0)

{

printf("\nThe parent process is id is %d\n",getpid());

printf("\nThe parent's parent process id is %d",getppid());

printf("\nthis stmt will be executed by parent\n");

}

else if(pid==0)

{

printf("\nThis stmt will be executed by child\n");

printf("\nThe child process id is %d",getpid());

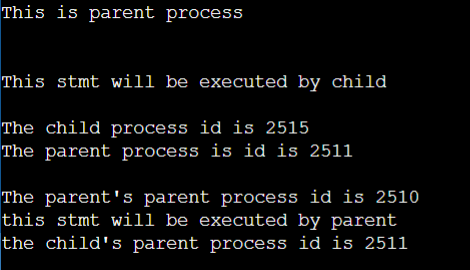
printf("\nthe child's parent process id is %d",getppid());

}

return 1;

}

**OUTPUT:**



**PROGRAM-4:**

**Aim:**

Process Management-Fork (), (Orphan and Zombie process) Exec(), Wait(), Sleep().

**Description:**

Process management is a crucial aspect of operating systems, encompassing various functions such as creating new processes, managing their execution, and handling their termination. Zombie processes are completed but remain in the process table until their parent processes retrieve their exit status. Orphan processes are those whose parent processes have terminated before them, leaving them to be adopted by the init process.

1. **fork():** 
   * **Purpose:** Creates a new process by duplicating the calling process.
   * **Syntax:** pid\_t fork(void); • **Example:**

#include <stdio.h> #include <unistd.h> int main() {

fork();

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

return 0;

}

1. **exec():** 
   * **Purpose:** Replaces the current process image with a new one.
   * **Syntax:** int exec(const char \*path, char \*const argv[]); • **Example:**

#include <stdio.h> #include <unistd.h> int main() {

char \*args[] = {"ls", "-l", NULL}; exec("/bin/ls", args); return 0;

}

1. **wait():** 
   * **Purpose:** Waits for a child process to terminate and returns its exit status.
   * **Syntax:** pid\_t wait(int \*status); • **Example:**

#include <stdio.h> #include <sys/wait.h> int main() {

int status; pid\_t pid = fork(); if (pid == 0) {

// Child process

sleep(3); return 42;

}

else if (pid > 0) {

// Parent process wait(&status);

printf("Child process exited with status: %d\n", WEXITSTATUS(status));

}

return 0;

}

**4. sleep():**

* **Purpose:** Suspends execution of the calling process for a specified number of seconds.
* **Syntax:** unsigned int sleep(unsigned int seconds); • **Example:**

#include <stdio.h> #include <unistd.h> int main() {

printf("Sleeping for 5 seconds...\n"); sleep(5); printf("Awake!\n"); return 0;

}

**PROGRAM 4**

**Q)Process Management- Fork (),(Orphan and Zombie process) Exec(),Wait(),Sleep()..**

**PROGRAM:**

**a)Wait()**

#include<stdio.h>

int main()

{

int pid=fork();

if(pid==0)

{

printf("CH: My pid is %d \n",getpid());

printf("CH: My parent pid id %d \n",getppid());

printf("\n CH: Inside child,it will terminate in no time \n");

}

else if(pid>0)

{

wait(10);

printf("this is parent,will go to sleep for 20 seconds \n");

printf("PT: My pid is %d \n",getpid());

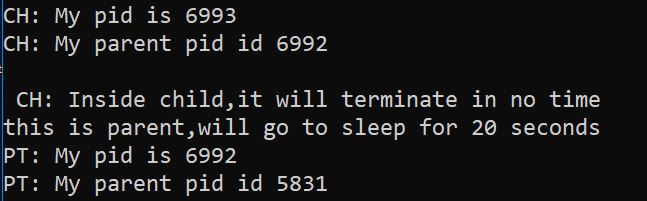
printf("PT: My parent pid id %d \n",getppid());

}

return 0;

}

**OUTPUT:**



**b)Sleep():**

#include<stdio.h>

int main()

{

int pid=fork();

if(pid==0)

{

printf("CH: My pid is %d \n",getpid());

printf("CH: My parent pid id %d \n",getppid());

printf("\n CH: Inside child,it will terminate in no time \n");

}

else if(pid>0)

{

printf("this is parent,will go to sleep for 20 seconds \n");

printf("PT: My pid is %d \n",getpid());

printf("PT: My parent pid id %d \n",getppid());

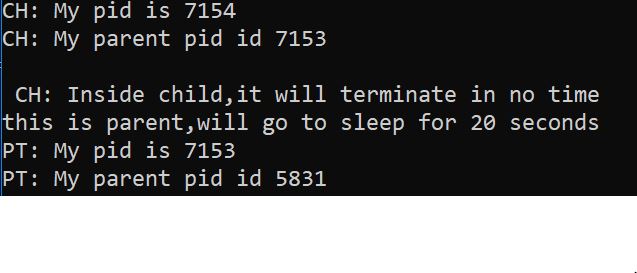
sleep(20);

}

return 0;

}

**OUTPUT:**



**PROGRAM-5:**

Program to demonstrate usage of Pipes, Shared memory, Message Queues.

**Description:**

* **Pipes:** 
  + - They can be used for communication between a parent process and its child process.
    - Pipes are implemented using the pipe() system call in Unix-like operating systems.
    - They have a limited capacity for data transmission and are suitable for small amounts of data.
    - Communication through pipes is typically achieved using file descriptors for reading from and writing to the pipe.
    - Pipes can be either anonymous (created using pipe()) or named (created using mkfifo()).
* **Shared Memory:** 
  + - Shared memory allows multiple processes to access the same region of memory concurrently.
    - It provides high-speed communication between processes since data is directly shared in memory.
    - Shared memory segments are created using system calls such as shmget(), shmat(), and shmctl() in Unix-like systems.
    - Synchronization mechanisms like semaphores or mutexes are often used to coordinate access to shared memory to prevent race conditions.
    - Shared memory is commonly used in scenarios where processes need to exchange large amounts of data quickly.
    - Cleanup of shared memory segments is essential to avoid memory leaks and can be done using shmctl() or other related system calls.
* **Message Queues:** 
  + - Message queues provide a communication mechanism between processes through message passing.
    - They allow processes to send and receive messages in a first-infirst-out (FIFO) manner.
    - Each message in a queue consists of a type identifier and data payload.
    - Message queues are implemented using system calls like msgget(), msgsnd(), and msgrcv() in Unix-like systems.
    - They can be used for inter-process communication in scenarios where processes need to exchange structured data.
    - Message queues offer more flexibility and robustness compared to pipes, especially for communication between unrelated processes.

**PROGRAM 5**

**Program to demonstrate usage of Pipes, Shared memory, Message Queues.**

**PROGRAM:**

**a)PIPES:**

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<string.h>

#define MSGSIZE 5

int main()

{

int pd[2];

pipe(pd);

int pid=fork();

if(pid==0)

{

printf("Child PID is %d\n",getpid());

printf("Read and write descriptors of the child process are %d,

%d\n",pd[0],pd[1]);

char buff[MSGSIZE];

close(pd[1]);

read(pd[0],buff,MSGSIZE);

buff[MSGSIZE]='\0';

printf("The message read by the child process is %s\n",buff);

close(pd[0]);

close(pd[1]);

}

else

{

printf("Read and write descriptors of the parent process are %d,

%d\n",pd[0],pd[1]);

char \*str="Hello";

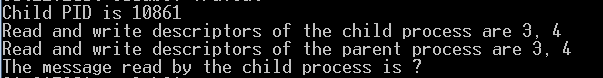
close(pd[0]);

close(pd[1]);

}

}

**OUTPUT:**



**b)MESSAGE QUEUE:**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/msg.h>

#include<string.h>

int main()

{

int msqid=msgget((key\_t)62,IPC\_CREAT|0644);

if(msqid<0)

{

printf("Unable to create msg");

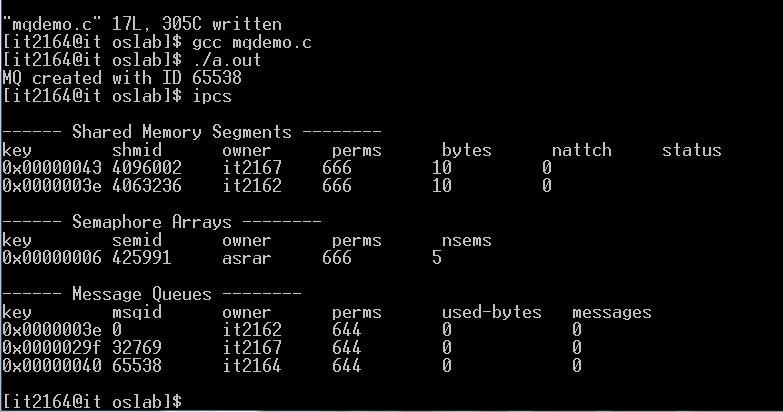
exit(1);

}

printf("MQ created with ID %d",msqid);

}

**OUTPUT:**



**c)SHARED MEMORY:**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/ipc.h>

int main()

{

int shmid=shmget((key\_t)62,10,IPC\_CREAT|0666);

printf("Shared memory created with id %d\n",shmid);

char\*ptr=(char\*)shmat(shmid,0,0);

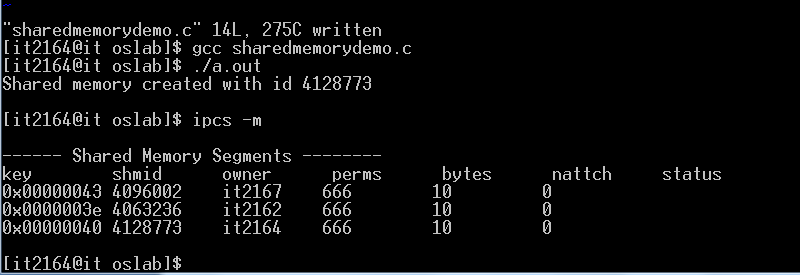
read(0,ptr,5);

write(1,ptr,5);

return 1;

}

**OUTPUT:**



**PROGRAM-6:**

Program for echo server using pipes.

**Description:**

* **Echo Server:** 
  + - Echo servers simply receive data from clients and send back the same data, facilitating bi-directional communication.
    - They are often used for learning network programming due to their straightforward implementation using socket programming.
    - Echo servers provide a basic example of server-client interaction and are commonly used for testing network connections.
* **Pipes:** 
  + They can be used for communication between a parent process and its child process.
  + Pipes are implemented using the pipe() system call in Unix-like operating systems.
  + They have a limited capacity for data transmission and are suitable for small amounts of data.
  + Communication through pipes is typically achieved using file descriptors for reading from and writing to the pipe.
  + Pipes can be either anonymous (created using pipe()) or named (created using mkfifo()).

**Algorithm:**

* + Create a pipe using the pipe() system call.
  + Fork the process to create a child process.
  + In the child process, close the write end of the pipe and execute the echo server logic.
  + In the parent process, close the read end of the pipe and execute the client logic.
  + In the echo server logic, continuously read data from the pipe using read() and write it back to the pipe using write().
  + In the client logic, prompt the user for input, write it to the pipe using write(), and read the echoed response from the pipe using read().
  + Close the write end of the pipe in the child process and the read end of the pipe in the parent process to release system resources.

**PROGRAM 6**

**Q)Program for echo server using pipes**

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

int main()

{

int pd1[2],pd2[2];

pipe(pd1);

pipe(pd2);

printf("pipe 1 read and write descriptors are %d,%d\n",pd1[0],pd1[1]);

printf("pipe 2 read and write descriptors are %d,%d\n",pd2[0],pd2[1]);

int pid=fork();

if(pid==0)

{

close(pd1[0]);

close(pd2[1]);

printf("read and write descriptors in client are %d,%d\n",pd2[0]

,pd1[1]);

char \*msg="Hello";

write(pd1[1],msg,5);

printf("Message written to pipe 1 by client %s\n",msg);

char buf[5];

read(pd2[0],buf,5);

printf("the echoed message from server to client is %s\n",buf);

close(pd1[1]);

close(pd2[0]);

printf("client is terminating\n");

}

else if(pid>0)

{

close(pd1[1]);

close(pd2[0]);

printf("Read and write descriptor in server are %d,%d\n",pd1[0],

pd2[1]);

char buf[5];

read(pd1[0],buf,5);

printf("the message received from child is %s\n",buf);

write(pd2[1],buf,5);

printf("Message echoed back to client by pipe 2 %s\n",buf);

close(pd1[0]);

close(pd2[1]);

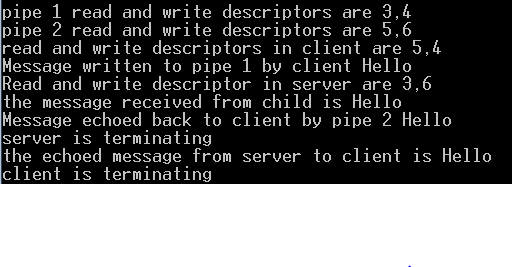
printf("server is terminating\n");

}

return 1;

}

**OUTPUT:**



**PROGRAM-7:**

Program for echo server using messages queues.

**Description:**

* **Message Queues:** 
  + - Message queues provide a communication mechanism between processes, allowing them to exchange structured messages in a first-in-first-out (FIFO) manner.
    - Each message in a queue consists of a type identifier and a data payload, facilitating the transmission of complex data structures between processes.
    - Message queues are commonly used in inter-process communication scenarios where processes need to exchange data asynchronously, ensuring robustness and flexibility in communication patterns.
* **Pipes:** 
  + They can be used for communication between a parent process and its child process.
  + Pipes are implemented using the pipe() system call in Unix-like operating systems.
  + They have a limited capacity for data transmission and are suitable for small amounts of data.
  + Communication through pipes is typically achieved using file descriptors for reading from and writing to the pipe.
  + Pipes can be either anonymous (created using pipe()) or named (created using mkfifo()).

**Algorithm:**

* + Create a message queue for communication between the echo server and clients using msgget() system call.
  + Fork the process to create a child process for handling client connections.
  + In the child process, set up a message queue to receive messages from clients.
  + Continuously listen for messages from clients using msgrcv() system call.
  + Upon receiving a message, extract the data payload and send it back to the client by placing it in a new message using msgsnd() system call.
  + In the parent process, set up a message queue to send messages to the echo server.
  + Prompt the user for input and send the message to the echo server using msgsnd() system call.
  + Continuously listen for messages from the echo server using msgrcv() system call.
  + Upon receiving a message, display it to the user.
  + Properly handle errors for system calls like msgget(), msgrcv(), and msgsnd().
  + Cleanup by removing the message queues using msgctl() system call when the server is terminated.

**PROGRAM 7**

**Q)Program for echo server using message queues**

**PROGRAM:**

#include<unistd.h>

#include<stdio.h>

#include<sys/types.h>

#include<sys/msg.h>

#include<string.h>

struct msgbuf

{

long mtype;

char mtext[50];

} m1,m2;

main()

{

int msqid,pid;

msqid=msgget((key\_t)67,IPC\_CREAT | 0666);

pid=fork();

if(pid==0)

{

printf("Child Process\n");

m1.mtype=1;

strcpy(m1.mtext,"hello");

msgsnd(msqid,(struct msgbuf \*)&m1,sizeof(m1),0);

sleep(2);

printf("In Child process again \n");

msgrcv(msqid,(struct msgbuf \*)&m2,sizeof(m2),1,0);

printf("msg recvd from parent in child process is %s \n",m2.mtext);

}

else

{

printf("ParentProcess\n");

msgrcv(msqid,(struct msgbuf \*)&m2,sizeof(m2),1,0);

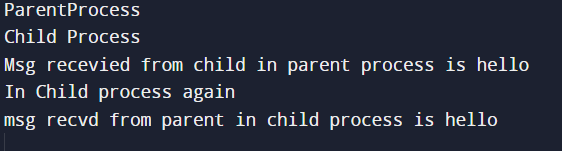
printf("Msg recevied from child in parent process is %s\n",m2.mtext);

msgsnd(msqid,(struct msgbuf\*)&m2,sizeof(m2),0);

}

}

**OUTPUT:**



**PROGRAM-8:**

Program for echo server using shared memory.

**Description:**

* **Shared Memory:** 
  + - Shared memory allows multiple processes to access the same region of memory concurrently, enabling efficient communication and data sharing between processes.
    - It provides high-speed communication between processes since data is directly shared in memory, without the need for copying or serialization.
    - Shared memory segments are created using system calls such as shmget(), shmat(), and shmctl() in Unix-like systems, with synchronization mechanisms like semaphores or mutexes used to coordinate access and prevent race conditions.
* **Pipes:** 
  + They can be used for communication between a parent process and its child process.
  + Pipes are implemented using the pipe() system call in Unix-like operating systems.
  + They have a limited capacity for data transmission and are suitable for small amounts of data.
  + Communication through pipes is typically achieved using file descriptors for reading from and writing to the pipe.
  + Pipes can be either anonymous (created using pipe()) or named (created using mkfifo()).

**Algorithm:**

* Create a shared memory segment using shmget() system call.
  + Attach the shared memory segment to the server process using shmat() system call.
  + Fork the process to create a child process for handling client connections.
  + In the child process, attach the shared memory segment to access the message queue.
  + Continuously listen for messages from clients in the shared memory segment.
  + Upon receiving a message, extract the data payload and send it back to the client by writing to the shared memory segment.
  + In the parent process, attach the shared memory segment to send messages to the echo server.
  + Prompt the user for input and write the message to the shared memory segment.
  + Continuously listen for messages from the echo server in the shared memory segment.
  + Upon receiving a message, display it to the user.
  + Properly handle errors for system calls like shmget(), shmat(), and shared memory operations.
  + Cleanup by detaching the shared memory segment using shmdt() system call when the server is terminated.

**PROGRAM 8**

**Q)Program for echo server using shared Memory**

**PROGRAM:**

#include<stdio.h>

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/msg.h>

#include<string.h>

#include<stdlib.h>

main()

{

int shmid,pid;

char \*ptr;

shmid=shmget((key\_t)87,10,IPC\_CREAT | 0666);

ptr=(char \*) shmat(shmid,0,0);

pid=fork();

if(pid==0)

{

printf("\n CP \n");

strcpy(ptr,"CP-Welcome");

sleep(2);

write(1,ptr,10);

printf("\n");

shmctl(ptr,IPC\_RMID,0);

exit(0);

}

else

{

sleep(1);

printf("\n PP \n");

write(1,ptr,10);

printf("\n");

strcpy(ptr,"PP-WELCOME");

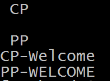
wait(0);

//exit(0);

}

}

**OUTPUT:**



**PROGRAM-9:**

Program for producer consumer problem using semaphores.

**Description:**

The producer-consumer problem involves two types of processes: producers and consumers. Producers generate data and place it into a shared buffer, while consumers retrieve and process the data from the buffer. The challenge is to ensure synchronization between producers and consumers to prevent issues such as buffer overflow or underflow. Semaphores are commonly used to coordinate access to the shared buffer, ensuring that producers wait when the buffer is full and consumers wait when the buffer is empty, thereby maintaining data integrity.

* **Semaphores:** 
  + Semaphores are synchronization primitives used in concurrent programming to control access to shared resources among multiple processes or threads.
  + They act as counters that are incremented and decremented by processes or threads to indicate resource availability or usage.
  + Semaphores provide mechanisms for process synchronization and mutual exclusion, allowing processes to coordinate their activities and prevent race conditions when accessing shared resources.

**Algorithm:**

* + Initialize semaphores for empty and full slots in the buffer.
  + Initialize the shared buffer.
  + Create producer and consumer processes.

In the producer process:

* + - * Wait on the empty semaphore to indicate space in the buffer.
      * Add an item to the buffer.
      * Signal the full semaphore to indicate the addition of an item.
* In the consumer process:
  + Wait on the full semaphore to indicate available items in the buffer.
  + Retrieve an item from the buffer.
  + Signal the empty semaphore to indicate consumption of an item.
* Repeat steps 4 and 5 indefinitely in both producer and consumer processes.
* Properly handle synchronization and mutual exclusion using semaphores to prevent race conditions.
* Ensure proper termination of the processes and cleanup of resources when the program ends.

**PROGRAM 9**

**Q)Program for producer consumer problem using semaphores**

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

int mutex=1,full=0,empty,x=0;

main()

{

int n,size;

void producer();

void consumer();

int wait(int);

int signal(int);

printf("Enter the size :");

scanf("%d",&size);

empty=size;

printf("\n1.producer\n2.consumer\n3.exit\n");

while(1)

{

printf("\nEnter your choice : ");

scanf("%d",&n);

switch(n)

{

case 1: if((mutex==1)&&(empty!=0))

producer();

else

printf("buffer is full\n");

break;

case 2: if((mutex==1)&&(full!=0))

consumer();

else

printf("buffer is empty\n");

break;

case 3: exit(0);

}

}

}

void producer()

{

mutex=wait(mutex);

full=signal(full);

empty=wait(empty);

x++;

printf("producer produces the item %d\n",x);

mutex=signal(mutex);

}

void consumer()

{

mutex=wait(mutex);

full=wait(full);

empty=signal(empty);

printf("consumer consumes the item %d\n",x);

x--;

mutex=signal(mutex);

}

int wait(s)

{

return (--s);

}

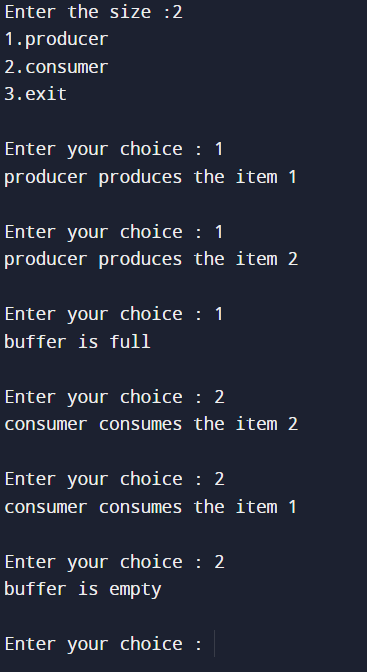
int signal(s)

{

return (++s);

}

**OUTPUT:**



**PROGRAM-10:**

Program for dining philosopher problem using semaphores.

**Description:**

The dining philosophers problem is a classic synchronization challenge where a group of philosophers sits around a table, each with a fork between them. To eat, a philosopher needs both adjacent forks. The problem arises when all philosophers try to pick up one fork simultaneously, leading to potential deadlock. The goal is to design a solution that prevents deadlock and starvation, ensuring each philosopher can eat without waiting indefinitely for access to forks. Various synchronization techniques, such as semaphores or monitors, are used to tackle this problem efficiently.

❖ **Semaphores:**

* Semaphores are synchronization primitives used in concurrent programming to control access to shared resources among multiple processes or threads.
* They act as counters that are incremented and decremented by processes or threads to indicate resource availability or usage.
* Semaphores provide mechanisms for process synchronization and mutual exclusion, allowing processes to coordinate their activities and prevent race conditions when accessing shared resources.

**Algorithm:**

* Initialize semaphores for each fork and a mutex semaphore to ensure exclusive access to shared resources.

Create philosopher processes, each representing a philosopher at the table.

* In each philosopher process:
  + Enter an infinite loop to represent the philosopher's dining behavior.
  + Think for a random amount of time.
  + Acquire the mutex semaphore to prevent multiple philosophers from accessing shared resources simultaneously.
  + Attempt to acquire the left and right forks using their respective semaphores.
  + If successful in acquiring both forks, eat for a random amount of time.
  + Release the acquired forks by signaling their respective semaphores.
  + Release the mutex semaphore to allow other philosophers to access shared resources.
* Repeat steps 3a to 3g indefinitely for each philosopher process.
* Properly handle synchronization and mutual exclusion using semaphores to prevent deadlocks and ensure fair access to forks.
* Ensure proper termination of the processes and cleanup of resources when the program ends.

**PROGRAM 10**

**Q)Program for dining philosopher problem using semaphores**

**PROGRAM:**

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/sem.h>

#include <stdio.h>

struct sembuf sop;

int chopstick;

main()

{

chopstick = semget((key\_t)77, 5, IPC\_CREAT | 0666);

semctl(chopstick, 0, SETVAL, 1);

semctl(chopstick, 1, SETVAL, 1);

semctl(chopstick, 2, SETVAL, 1);

semctl(chopstick, 3, SETVAL, 1);

semctl(chopstick, 4, SETVAL, 1);

if (fork() == 0)

philospher(0);

else if (fork() == 0)

{

philospher(1);

sleep(5);

}

else if (fork() == 0)

{

philospher(2);

sleep(5);

}

else if (fork() == 0)

{

philospher(3);

sleep(5);

}

else

{

philospher(4);

sleep(5);

}

}

int philospher(int i)

{

while (1)

{

if (i == 0 || i == 2 || i == 4)

{

printf("\n philospher %d is thinking\n", i);

sleep(5);

down(chopstick, i);

down(chopstick, (i + 1) % 5);

printf("\n philospher %d has acquired chopstick\n", i);

sleep(5);

printf("\n philospher %d is eating\n", i);

sleep(5);

up(chopstick, i);

up(chopstick, (i + 1) % 5);

printf("\n philospher %d has returned chopstick\n", i);

sleep(5);

}

else

{

printf("\n philospher %d is thinking\n", i);

sleep(5);

down(chopstick, (i + 1) % 5);

down(chopstick, i);

printf("\n philospher %d has acquired chopstick\n", i);

sleep(5);

printf("\n philospher %d is eating\n", i);

sleep(5);

up(chopstick, (i + 1) % 5);

up(chopstick, i);

printf("\n philospher %d has retained chopstick\n", i);

sleep(5);

}

}

}

int down(int semid, int num)

{

sop.sem\_num = num;

sop.sem\_op = -1;

sop.sem\_flg = 0;

semop(semid, &sop, 1);

}

int up(int semid, int num)

{

sop.sem\_num = num;

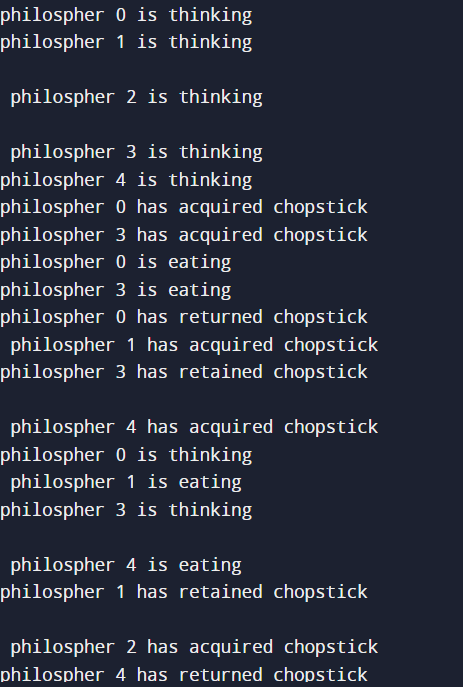
sop.sem\_op = +1;

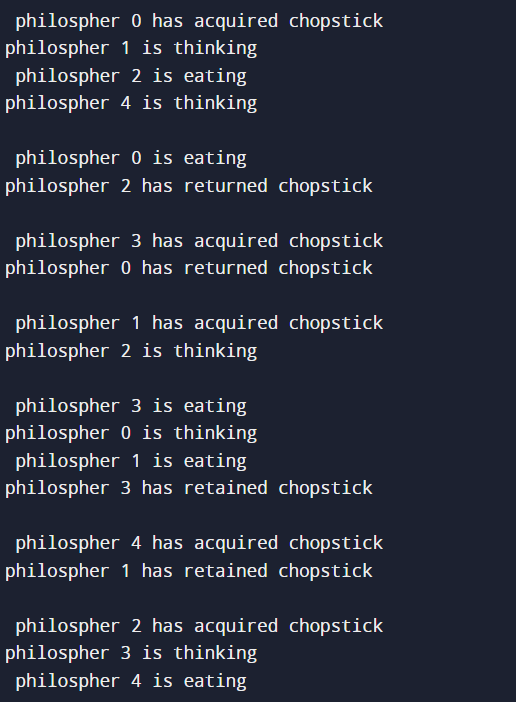
sop.sem\_flg = 0;

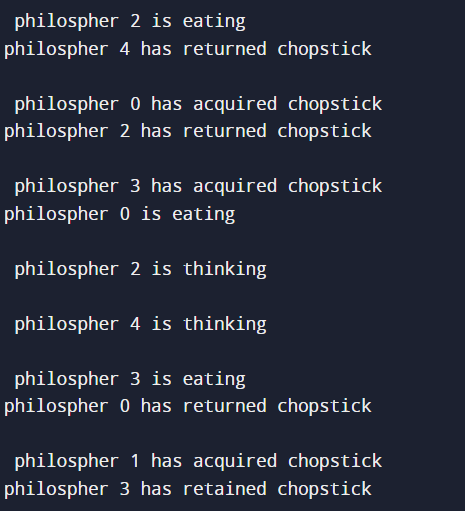
semop(semid, &sop, 1);

}

**OUTPUT:**







**PROGRAM-11:**

Program to implement Processor Scheduling Algorithms: a) FCFS b) SJF

**Description:**

Processor scheduling algorithms are used in operating systems to manage the execution of multiple processes on a single processor. They determine the order in which processes are selected for execution, optimizing factors such as turnaround time, response time, and throughput. Shortest Job First (SJF) scheduling selects the process with the shortest burst time first, minimizing average waiting time and maximizing throughput. First-Come, First-Served (FCFS) scheduling, on the other hand, executes processes in the order they arrive, which is simple and fair but may result in long average waiting times, especially for long processes.

* **Shortest Job First (SJF):** 
  + SJF scheduling minimizes average waiting time by selecting the process with the shortest burst time first, leading to better turnaround time and higher throughput.
  + It may result in starvation for long processes if shorter processes constantly arrive, as they will always be scheduled first.
  + SJF can be either preemptive or non-preemptive, with preemptive SJF allowing shorter jobs to interrupt longer ones currently executing.
* **First-Come, First-Served (FCFS):** 
  + FCFS scheduling executes processes in the order they arrive, which is simple and fair but may result in long average waiting times, especially for long processes.
  + It does not require any additional data structures or bookkeeping, making it easy to implement and understand.
  + FCFS may not be suitable for time-sharing systems where responsiveness is crucial, as long processes can block short ones from executing, leading to poor overall system performance.

**Algorithm:**

* + **FCFS (First-Come, First-Served):**
  + Initialize a queue to hold the processes.
  + Add processes to the queue in the order they arrive.
  + Execute processes in the queue sequentially, starting with the first process added.
  + Wait for the current process to finish execution before moving on to the next process in the queue.
  + Continue executing processes until the queue is empty.

• **SJF (Shortest Job First):**

* Initialize a queue to hold the processes. o Add processes to the queue as they arrive.
* Sort the queue based on the burst time (execution time) of each process, with the shortest job at the front of the queue.
* Execute processes in the sorted queue sequentially, starting with the process with the shortest burst time.
* Wait for the current process to finish execution before moving on to the next process.

**PROGRAM 11**

**Q)Program to implement Processor Scheduling Algorithms a) FCFS b) SJF**

**PROGRAM:**

**a)FCFS**

#include<stdio.h>

main()

{

int n,i;

float avgwt=0.0,avgta=0.0;

printf("\nenter the no of process\n");

scanf("%d",&n);

int p[n],bt[n],wt[n],ta[n];

for(i=0;i<n;i++)

{

printf("\nenter the process no.\n");

scanf("%d",&p[i]);

printf("enter the cpu time for this process\n");

scanf("%d",&bt[i]);

}

wt[0]=0;

ta[0]=bt[0];

avgta=ta[0];

for(i=1;i<n;i++)

{

wt[i]=wt[i-1]+bt[i-1];

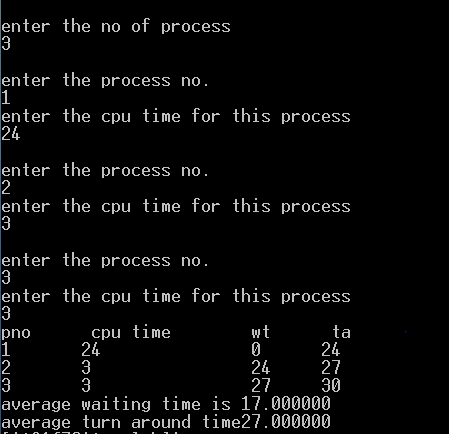
avgwt=avgwt+wt[i];

ta[i]=wt[i]+bt[i];

avgta=avgta+ta[i];

}

}

**OUTPUT:**

**b)SJFS**

#include<stdio.h>

main()

{

int n;

float avgturn=0,avgwait=0;

printf("\nenter the no of process");

scanf("%d",&n);

int wait[n],turntime[n],i,btime[n];

printf("rnter process burst time\n");

int temp,j;

for(i=0;i<n;i++)

{

scanf("%d",&btime[i]);

}

printf("sorting by burst time\n");

for(i=0;i<n;i++)

{

for(j=i+1;j<n;j++)

{

if(btime[j]<btime[i])

{

temp=btime[j];

btime[j]=btime[i];

btime[i]=temp;

}

}

}

wait[0]=0;

for(i=1;i<n;i++)

{

wait[i]=wait[i-1]+btime[i-1];

avgwait+=wait[i];

}

for(i=0;i<n;i++)

{

turntime[i]=wait[i]+btime[i];

avgturn+=turntime[i];

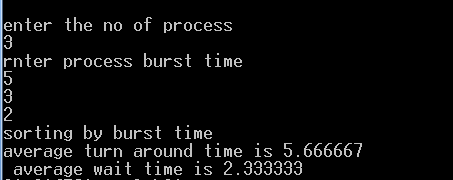
}

printf("average turn around time is %f\n average wait time is %f\n",(avg

turn/n),(avgwait/n));

}

**OUTPUT:**



**COMPUTER NETWORKS**

**PROGRAM-1:**

**Aim:**

Understanding and using the following Utility commands: ifconfig, netstat, ping, arp, telnet, ftp, finger.

**Description:**

These utility commands are essential for network troubleshooting, administration, and communication tasks in Unix-like operating systems.

1. **ifconfig (Interface Configuration):**
   * **Purpose:** ifconfig is used to view and configure network interfaces on a Unix-like operating system.
   * **Syntax:** ifconfig [interface] [options]
   * **Example:** ifconfig eth0 up (Brings up the Ethernet interface eth0)
2. **netstat (Network Statistics):**
   * **Purpose:** netstat is used to display network connections, routing tables, interface statistics, masquerade connections, and multicast memberships.
   * **Syntax:** netstat [options]
   * **Example:** netstat -a (Displays all active network connections)
3. **ping:**
   * **Purpose:** ping is used to test the reachability of a host on an Internet Protocol (IP) network and to measure the round-trip time for messages sent from the originating host to a destination computer.
   * **Syntax:** ping [destination] [options]

**Example:** ping 8.8.8.8 (Sends ICMP echo requests to Google's DNS server)

1. **arp (Address Resolution Protocol):**

* **Purpose:** arp is used to display and modify the IP-to-MAC address translation tables used by the Address Resolution Protocol (ARP).
* **Syntax: a**rp [options]
* **Example:** arp -a (Displays the current ARP table)

1. **telnet:**
   * **Purpose:** telnet is used to establish a command-line session on a remote host over a network connection, typically using the Telnet protocol.
   * **Syntax:** telnet [host] [port]
   * **Example:** telnet example.com 80 (Connects to the web server on example.com using port 80)
2. **ftp (File Transfer Protocol):**
   * **Purpose:** ftp is used to transfer files between a local computer and a remote computer over a network using the File Transfer Protocol.
   * **Syntax:** ftp [options] [host]
   * **Example:** ftp ftp.example.com (Connects to the FTP server on [ftp.example.com](ftp://ftp.example.com/))
3. **finger:**
   * **Purpose:** finger is used to retrieve information about users on a remote system, such as their login name, full name, terminal session status, and more.
   * **Syntax:** finger [username@]host
   * **Example:** finger user@example.com (Retrieves information about the user "user" on the host example.com)

**PROGRAM 1**

**Q)Understanding and using the following Utility commands: Ifconfig, netstat, ping, arp, telnet, ftp, finger**

**SOCKET PROGRAMMING:**

**PROGRAM-2:**

**Aim:**

Implementation of Connection-Oriented Server using Standard ports:

* + Echo Server (7)
  + Date and Time Server (13)
  + Character Generator (19)

**Description:**

Connection-oriented servers provide a reliable and ordered communication channel between clients and servers, offering features such as connection establishment, data integrity, and error recovery. They are widely used in networked applications where consistent and predictable communication is essential for successful operation.

**1. Echo Server (port 7):**

* + - **Description:** Echoes back data received from clients.
    - **Functionality:** When a client connects, the server sends back any data it receives.
    - **Use Case:** Troubleshooting and testing network connectivity.

**2. Date and Time Server (port 13):**

* + - **Description:** Provides the current date and time to clients.
    - **Functionality:** Clients request the current date and time, and the server responds.
    - **Use Case:** Timestamping and time synchronization in network applications.

**3. Character Generator (port 19):**

* + **Description:** Generates and sends random ASCII characters.

**Functionality:** Upon connection, the server continuously sends a stream of random characters.

* + **Use Case:** Network testing, security assessment, and cryptographic applications.

**Algorithm:**

* + Initialize a TCP socket and bind it to the specified port.
  + Enter a loop to continuously listen for incoming connection requests.
  + Accept incoming connections using the accept() function.
  + Upon accepting a connection, enter another loop to handle requests from the client.
  + Read data from the client using recv() and process it based on the service requested.
  + For the Echo Server (port 7), send back the received data to the client.
  + For the Date and Time Server (port 13), retrieve the current date and time, convert it to a string, and send it to the client.
  + For the Character Generator (port 19), generate random ASCII characters and send them to the client.
  + Return to the loop to continue handling requests.
  + Close the connection with the client when communication is complete or terminated.
  + Close the listening socket when the server is terminated.

**PROGRAM 2**

**Q)Implementation of Connection-Oriented Server using Standard ports**

**i. Echo Server (7)**

#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<stdlib.h>

#include<string.h>

int main()

{

char smsg[30],rmsg[30];

struct sockaddr\_in ser;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(7);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_STREAM,IPPROTO\_TCP);

int rval=connect(sockfd,(struct sockaddr\*)&ser,sizeof(ser));

printf("Client connection is established with echo server\n");

printf("Enter a msg to be sent to Echo server\n");

scanf("%s",&smsg);

send(sockfd,(char\*)smsg,strlen(smsg),0);

printf("Client has succesfully send the message to server\n");

recv(sockfd,(char\*)rmsg,26,0);

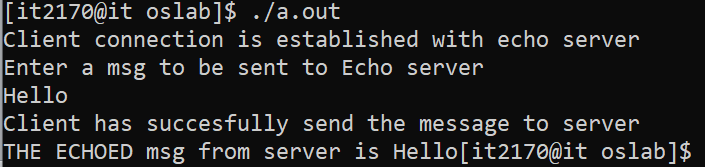
printf("THE ECHOED msg from server is %s",rmsg);

close(sockfd);

return 1;

}

**OUTPUT:**



**ii. Date and Time Server(13)**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

int main()

{

char msg[30];

struct sockaddr\_in ser;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(13);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_STREAM,IPPROTO\_TCP);

int rval=connect(sockfd,(struct sockaddr \*)&ser,sizeof(ser));

printf("Client Connection established with day time server\n");

recv(sockfd,(char \*)msg,26,0);

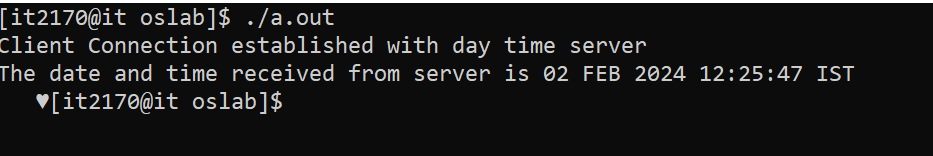
printf("The date and time received from server is %s",msg);

close(sockfd);

return 1;

}

**OUTPUT:**



**iii. Character Generation(19)**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

int main()

{

char msg[100];

struct sockaddr\_in ser;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(19);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_STREAM,IPPROTO\_TCP);

int rval=connect(sockfd,(struct sockaddr \*)&ser,sizeof(ser));

printf("Client Connection established with day time server\n");

recv(sockfd,(char \*)msg,72,0);

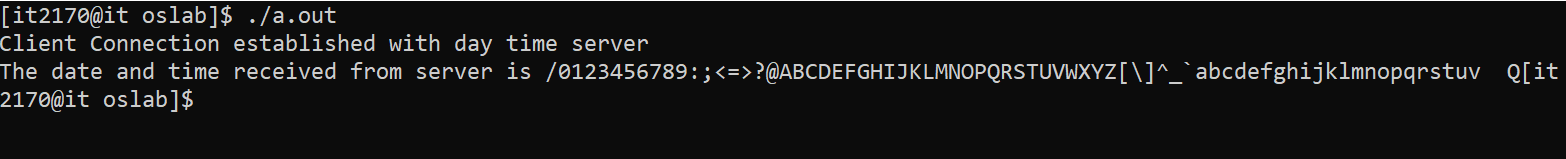
printf("The date and time received from server is %s",msg);

close(sockfd);

return 1;

}

**OUTPUT:**



**PROGRAM-3:**

Implementation of Connection-Less Server using Standard ports:

* + Echo Server (7)
  + Date and Time Server (13)
  + Character Generator (19)

**Description:**

Connection-less servers, also known as UDP (User Datagram Protocol) servers, operate without establishing a persistent connection between the server and clients. Instead, they use a fire-and-forget approach, where data packets are sent without prior communication or acknowledgment.

**1. Echo Server (port 7):**

* + - **Description:** Echoes back data received from clients.
    - **Functionality:** When a client connects, the server sends back any data it receives.
    - **Use Case:** Troubleshooting and testing network connectivity.

**2. Date and Time Server (port 13):**

* + - **Description:** Provides the current date and time to clients.
    - **Functionality:** Clients request the current date and time, and the server responds.
    - **Use Case:** Timestamping and time synchronization in network applications.

**3. Character Generator (port 19):**

* + **Description:** Generates and sends random ASCII characters.
  + **Functionality:** Upon connection, the server continuously sends a stream of random characters.

**Use Case:** Network testing, security assessment, and cryptographic applications.

**Algorithm:**

* + Initialize a UDP socket and bind it to the specified port.
  + Enter a loop to continuously listen for incoming UDP packets.
  + Upon receiving a packet, extract its data payload and the client's address and port.
  + Based on the port number, determine the requested service.
  + For the Echo Server (port 7), send back the received data to the client.
  + For the Date and Time Server (port 13), retrieve the current date and time, convert it to a string, and send it to the client.
  + For the Character Generator (port 19), generate random ASCII characters and send them to the client.
  + Repeat the loop to handle more requests.
  + Close the UDP socket when the server is terminated.

**PROGRAM 3**

**Q) Implementation of Connection-Less Server using Standard ports.**

**a) Character Generation(19):**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

int main()

{

char smsg[30],rmsg[130];

struct sockaddr\_in ser,cli;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(19);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

cli.sin\_family=AF\_INET;

cli.sin\_port=htons(5005);

cli.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_DGRAM,IPPROTO\_UDP);

int rval=bind(sockfd,(struct sockaddr\*)&cli,sizeof(cli));

printf("Client Binded successfully \n");

//printf("Enter a message\n");

//scanf("%s",&smsg);

int slen;

sendto(sockfd,smsg,5,0,(struct sockaddr\*)&ser,sizeof(ser));

recvfrom(sockfd,rmsg,72,0,(struct sockaddr\*)&ser,&slen);

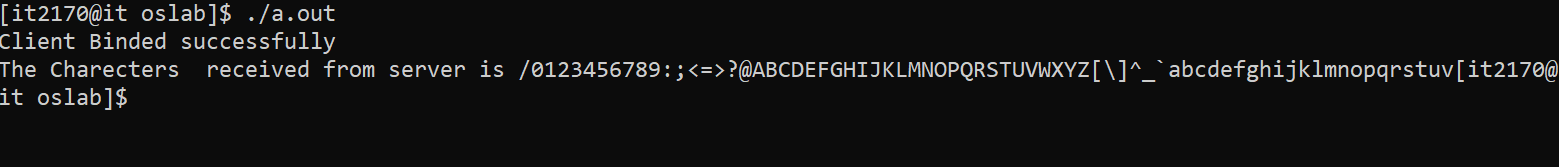
printf("The Charecters received from server is %s",rmsg);

close(cli);

return 1;

}

**OUTPUT:**



**b) Date and Time Server(13):**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

int main()

{

char smsg[30],rmsg[30];

struct sockaddr\_in ser,cli;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(13);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

cli.sin\_family=AF\_INET;

cli.sin\_port=htons(5005);

cli.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_DGRAM,IPPROTO\_UDP);

int rval=bind(sockfd,(struct sockaddr\*)&cli,sizeof(cli));

printf("Client Binded successfully \n");

//printf("Enter a message\n");

//scanf("%s",&smsg);

int slen;

sendto(sockfd,smsg,5,0,(struct sockaddr\*)&ser,sizeof(ser));

recvfrom(sockfd,rmsg,26,0,(struct sockaddr\*)&ser,&slen);

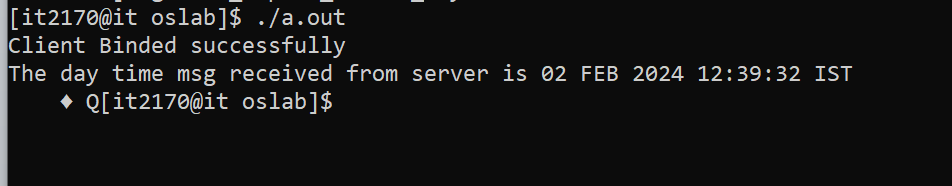
printf("The day time msg received from server is %s",rmsg);

close(cli);

return 1;

}

**OUTPUT:**



**c) Echo Service(7):**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

int main()

{

char smsg[30],rmsg[30];

struct sockaddr\_in ser,cli;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(7);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

cli.sin\_family=AF\_INET;

cli.sin\_port=htons(5005);

cli.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_DGRAM,IPPROTO\_UDP);

int rval=bind(sockfd,(struct sockaddr\*)&cli,sizeof(cli));

printf("Client Binded successfully \n");

printf("Enter a message\n");

scanf("%s",&smsg);

int slen;

sendto(sockfd,smsg,5,0,(struct sockaddr\*)&ser,sizeof(ser));

recvfrom(sockfd,rmsg,5,0,(struct sockaddr\*)&ser,&slen);

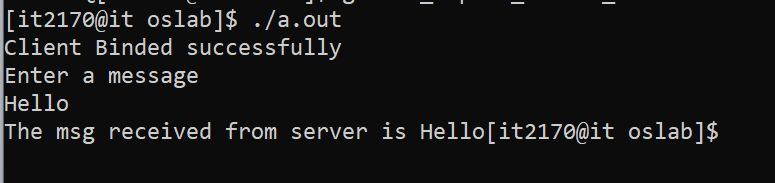
printf("The msg received from server is %s",rmsg);

close(cli);

return 1;

}

**OUTPUT:**



**PROGRAM-4:**

Implementation of Connection-Oriented Iterative Echo-server, Date and time, using User-defined ports:

**Description:**

An iterative server is a type of server that handles one client connection at a time, sequentially processing requests from clients. In the case of a connection-oriented iterative server, it operates using TCP (Transmission

Control Protocol) instead of UDP (User Datagram Protocol). Unlike UDP, TCP establishes a connection before sending data and guarantees reliable, ordered delivery of packets.

**1. Echo Server (port 7):**

* + - **Description:** Echoes back data received from clients.
    - **Functionality:** When a client connects, the server sends back any data it receives.
    - **Use Case:** Troubleshooting and testing network connectivity.

**1. Date and Time Server (port 13):**

* + **Description:** Provides the current date and time to clients.
  + **Functionality:** Clients request the current date and time, and the server responds.
  + **Use Case:** Timestamping and time synchronization in network applications.

**Algorithm:**

* + Initialize a TCP socket.
  + Bind the socket to a user-defined port for the desired service
  + Enter a loop to continuously listen for incoming connection requests.
  + Accept incoming connections using the accept() function.
  + Upon accepting a connection, enter another loop to handle requests from the client.
  + Read data from the client using recv() and send it back to the client using send().
  + For the Date and Time service, retrieve the current date and time, convert it to a string, and send it to the client.
  + Return to the loop to continue handling requests.
  + Close the connection with the client when communication is complete or terminated.
  + Close the listening socket when the server is terminated.

**PROGRAM 4**

**Q)implementation of connection oriented iterative using user-defined ports**

**a)echo-server(7)**

**client.c**

#include<stdio.h>

#include<sys/types.h>

#include<stdlib.h>

#include<sys/socket.h>

#include<netinet/in.h>

int main()

{

// port used is 5005 for echo service

struct sockaddr\_in ser;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(5005);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_STREAM,0);

int rval=connect(sockfd,(struct sockaddr\*)&ser,sizeof(ser));

printf(“CONECTION ESTBLISHED..\n”);

char smsg[20],rmsg[20];

printf(“Enter Message..\n”);

scanf("%s",&smsg);

send(sockfd,(char \*)smsg,5,0);

recv(sockfd,(char \*)rmsg,5,0);

printf("Echoed message received from server is %s",rmsg);

close(sockfd);

return 1;

}

**Server.c**

#include<stdio.h>

#include<sys/types.h>

#include<stdlib.h>

#include<sys/socket.h>

#include<netinet/in.h>

int main()

{

//running server echo srvice at port no 5005

struct sockaddr\_in ser,cli;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(5005);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_STREAM,0);

int rval=bind(sockfd,(struct sockaddr\*)&ser,sizeof(ser));

if(rval<0)

{

printf("BINDING ERROR");

exit(0);

}

listen(sockfd,5);

char msg[20];

int cli\_len;

while(1)

{

printf("Server is waiting for Active Connection Request from Client..\n");

int nsockfd=accept(sockfd,(struct sockaddr\* )&cli,&cli\_len);

recv(nsockfd,(char\*)msg,5,0);

send(nsockfd,(char\*)msg,5,0);

printf("Message Echoed to Client.., Closing the Connection..\n");

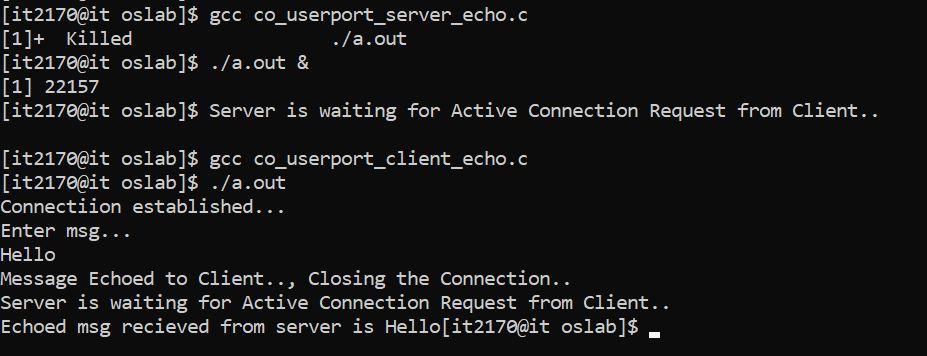
close(nsockfd);

}

return 1;

}

**OUTPUT:**



**b)date and time(13)**

**client.c**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

int main()

{

char msg[30];

struct sockaddr\_in ser;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(5006);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_STREAM,IPPROTO\_TCP);

int rval=connect(sockfd,(struct sockaddr \*)&ser,sizeof(ser));

printf("Client Connection established with day time server\n");

recv(sockfd,(char \*)msg,26,0);

printf("The date and time received from server is %s",msg);

close(sockfd);

return 1;

}

**Server.c**

#include<stdio.h>

#include<sys/types.h>

#include<stdlib.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<fcntl.h>

int main()

{

//running server daytime service at port no 5006

struct sockaddr\_in ser,cli;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(5006);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_STREAM,0);

int rval=bind(sockfd,(struct sockaddr\*)&ser,sizeof(ser));

if(rval<0)

{

printf("BINDING ERROR");

exit(0);

}

listen(sockfd,5);

char smsg[30],rmsg[30];

int cli\_len;

while(1)

{

printf("Server is waiting for Active Connection Request from Client..\n");

int nsockfd=accept(sockfd,(struct sockaddr\* )&cli,&cli\_len);

printf("Connection Request Arrived from client and Connection established successfully...\n");

system("date > f1");

int fd=open("f1",O\_RDONLY);

read(fd,rmsg,25);

close(fd);

rval=send(nsockfd,rmsg,sizeof(rmsg),0);

if(rval<0)

{

perror("send error:");

exit(0);

}

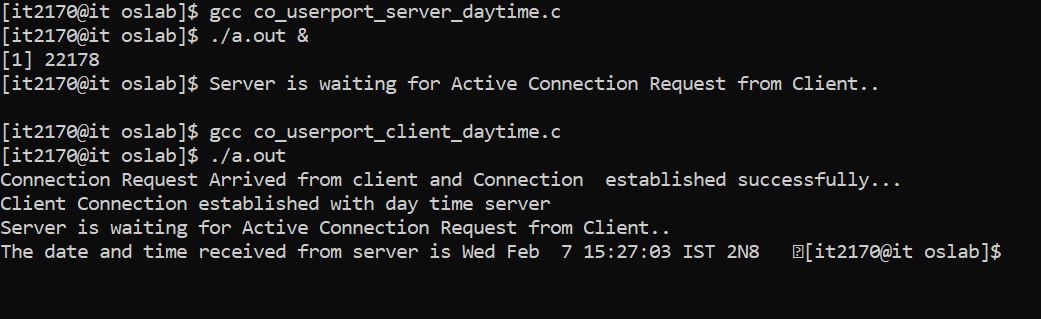
close(nsockfd);

}

return 1;

}

**OUTPUT:**



**c)character generation(19)**

**client.c**

#include<stdio.h>

#include<stdlib.h>

#include<sys/socket.h>

#include<sys/types.h>

#include<netinet/in.h>

int main()

{

char msg[100];

struct sockaddr\_in ser;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(5007);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_STREAM,0);

int rval=connect(sockfd,(struct sockaddr\*)&ser,sizeof(ser));

printf("THe connection Established with server....\n");

recv(sockfd,(char\*)msg,72,0);

printf("The Character generated from server is %s\n",msg);

return 0;

}

**Server.c**

#include<stdio.h>

#include<sys/types.h>

#include<stdlib.h>

#include<sys/socket.h>

#include<netinet/in.h>

int main()

{

//running server echo srvice at port no 5007

struct sockaddr\_in ser,cli;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(5007);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_STREAM,0);

int rval=bind(sockfd,(struct sockaddr\*)&ser,sizeof(ser));

if(rval<0)

{

printf("BINDING ERROR");

exit(0);

}

listen(sockfd,5);

char msg[120];

int cli\_len;

while(1)

{

printf("Server is waiting for Active Connection Request from Client..\n");

int nsockfd=accept(sockfd,(struct sockaddr\* )&cli,&cli\_len);

int i,j;

for(i=48,j=0;i<=119;i++,j++)

{

msg[j]=i;

}

msg[j]='\0';

send(nsockfd,(char\*)msg,72,0);

printf("Message Send to Client.., Closing the Connection..\n");

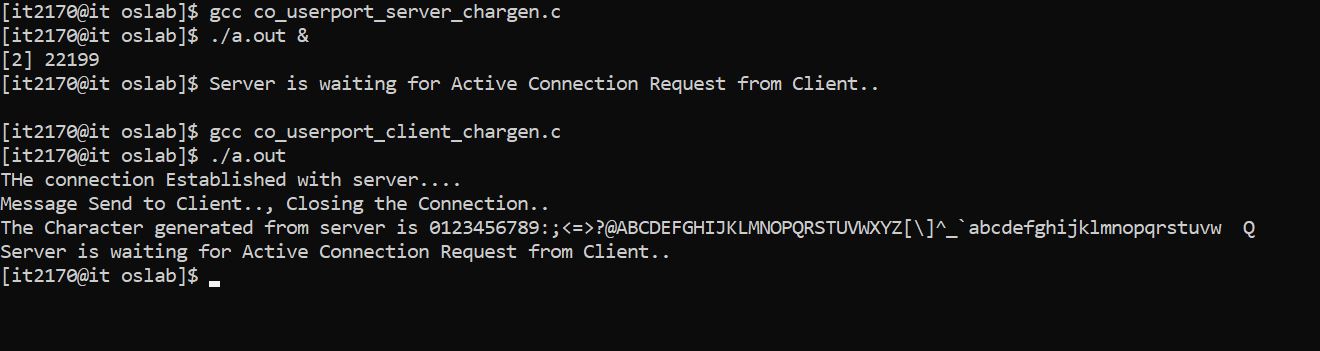
close(nsockfd);

}

return 1;

}

**OUTPUT:**



**PROGRAM-5:**

Implementation of Connection-Less Iterative Echo-server, Date and time, using User-defined ports:

**Description:**

An iterative server is a type of server that handles one client connection at a time, sequentially processing requests from clients. In the case of a connection-less iterative server, it operates using UDP (User Datagram Protocol) instead of TCP (Transmission Control Protocol). Unlike TCP, UDP does not establish a connection before sending data and does not guarantee delivery or order of packets.

**1. Echo Server (port 7):**

* + - **Description:** Echoes back data received from clients.
    - **Functionality:** When a client connects, the server sends back any data it receives.
    - **Use Case:** Troubleshooting and testing network connectivity.

**2. Date and Time Server (port 13):**

* + **Description:** Provides the current date and time to clients.
  + **Functionality:** Clients request the current date and time, and the server responds.
  + **Use Case:** Timestamping and time synchronization in network applications.

**Algorithm:**

* + Initialize a UDP socket.
  + Bind the socket to a user-defined port for the desired service.
  + Enter a loop to continuously listen for incoming UDP packets.

Upon receiving a packet, extract its data payload and the client's address and port.

* + Process the data payload based on the service requested (Echo, Date and Time).
  + For the Echo service, send back the received data to the client.
  + For the Date and Time service, retrieve the current date and time, convert it to a string, and send it to the client.
  + Return to the loop to continue listening for incoming packets.

Close the socket when the server is terminated.

**PROGRAM 5**

**Q)implementation of connectionless iterative using user-defined ports**

**a)echo-server(7)**

**client.c**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

int main()

{

char smsg[30],rmsg[30];

struct sockaddr\_in ser,cli;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(50011);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

cli.sin\_family=AF\_INET;

cli.sin\_port=htons(6005);

cli.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_DGRAM,IPPROTO\_UDP);

int rval=bind(sockfd,(struct sockaddr\*)&cli,sizeof(cli));

if(rval<0)

{

perror("Client Bind Error..\n");

exit(1);

}

printf("Client Binded successfully \n");

printf("Enter a message\n");

scanf("%s",smsg);

int slen;

sendto(sockfd,smsg,5,0,(struct sockaddr\*)&ser,sizeof(ser));

recvfrom(sockfd,rmsg,5,0,(struct sockaddr\*)&ser,&slen);

printf("The msg received from server is %s",rmsg);

close(cli);

return 1;

}

**Server.c**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

int main()

{

int cl;

char rmsg[20];

struct sockaddr\_in ser,cli;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(50011);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

cl=sizeof(cli);

int sockfd=socket(AF\_INET,SOCK\_DGRAM,0);

int rval=bind(sockfd,(struct sockaddr\*)&ser,sizeof(ser));

if(rval<0)

{

printf("BINDING ERROR");

exit(0);

}

printf("BINDED SUCCESSFULLY....\n");

while(1)

{

printf("Server waiting for Client request...\n");

recvfrom(sockfd,rmsg,5,0,(struct sockaddr \*)&cli,&cl);

printf("Recieved msg from client as %s...\n",rmsg);

sendto(sockfd,rmsg,5,0,(struct sockaddr \*)&cli,cl);

printf("send msg to client as %s...\n",rmsg);

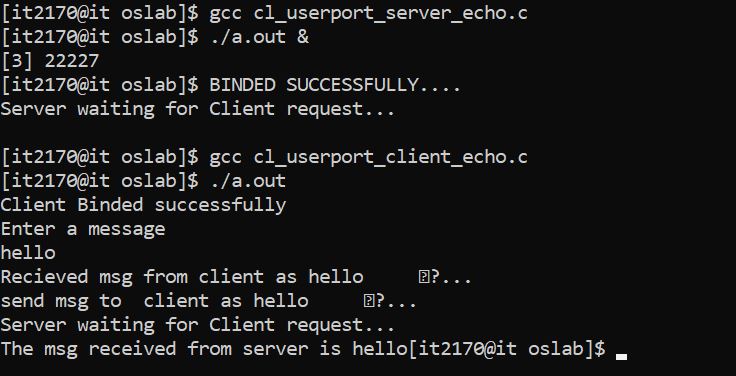
}

close(sockfd);

return 1;

}

**OUTPUT:**



**b)date and time(13)**

**client.c**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

int main()

{

char smsg[30],rmsg[30];

struct sockaddr\_in ser,cli;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(50012);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

cli.sin\_family=AF\_INET;

cli.sin\_port=htons(5005);

cli.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

int sockfd=socket(AF\_INET,SOCK\_DGRAM,IPPROTO\_UDP);

int rval=bind(sockfd,(struct sockaddr\*)&cli,sizeof(cli));

printf("Client Binded successfully \n");

//printf("Enter a message\n");

//scanf("%s",&smsg);

int slen;

sendto(sockfd,smsg,5,0,(struct sockaddr\*)&ser,sizeof(ser));

recvfrom(sockfd,rmsg,26,0,(struct sockaddr\*)&ser,&slen);

printf("The day time msg received from server is %s",rmsg);

close(cli);

return 1;

}

**server.c**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<fcntl.h>

int main()

{

int cl;

char rmsg[30];

struct sockaddr\_in ser,cli;

ser.sin\_family=AF\_INET;

ser.sin\_port=htons(50012);

ser.sin\_addr.s\_addr=inet\_addr("10.2.0.7");

cl=sizeof(cli);

int sockfd=socket(AF\_INET,SOCK\_DGRAM,0);

int rval=bind(sockfd,(struct sockaddr\*)&ser,sizeof(ser));

if(rval<0)

{

printf("BINDING ERROR");

exit(0);

}

printf("BINDED SUCCESSFULLY....\n");

while(1)

{

printf("Server waiting for Client request...\n");

recvfrom(sockfd,rmsg,5,0,(struct sockaddr \*)&cli,&cl);

//printf("Recieved msg from client as %s...\n",rmsg);

system("date>f1");

int fd=open("f1",O\_RDONLY);

read(fd,rmsg,26);

close(fd);

sendto(sockfd,rmsg,26,0,(struct sockaddr \*)&cli,cl);

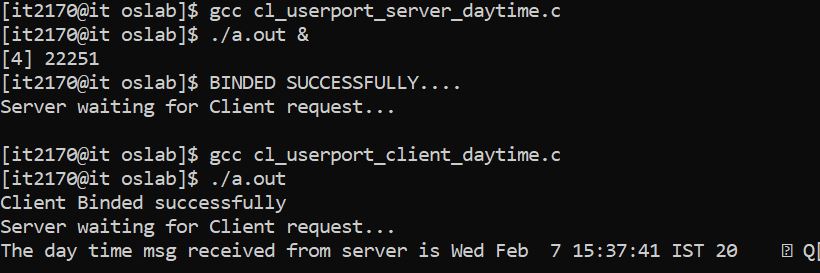
}

close(sockfd);

return 1;

}

**OUTPUT:**



**PROGRAM-6:**

Implementation of Connection-Oriented Concurrent Echo-server, Date and time, using User-defined ports.

**Description:**

In the case of a concurrent connection-oriented server, it operates using

TCP (Transmission Control Protocol) instead of UDP (User Datagram Protocol). Unlike UDP, TCP establishes concurrent connections with multiple clients, allowing the server to handle multiple client requests simultaneously while ensuring reliable, ordered delivery of packets..

**1. Echo Server (port 7):**

* + - **Description:** Echoes back data received from clients.
    - **Functionality:** When a client connects, the server sends back any data it receives.
    - **Use Case:** Troubleshooting and testing network connectivity.

**2. Date and Time Server (port 13):**

* + **Description:** Provides the current date and time to clients.
  + **Functionality:** Clients request the current date and time, and the server responds.
  + **Use Case:** Timestamping and time synchronization in network applications.

**Algorithm:**

* + Initialize TCP sockets for Echo, Date, and Time services.
  + Bind sockets to user-defined ports.
  + Listen for incoming connections on each socket.

Accept connections concurrently using threading or multiprocessing.

* + Handle client requests in separate threads or processes.
  + Read data from clients and process based on service.
  + Send response back to clients.
  + Repeat steps 5-7 for each client connection.
  + Close sockets and terminate threads/processes when server is terminated.

**PROGRAM 6**

**Q)Implementation of Connection-Oriented Concurrent Echo-server, Date and time, using User-defined ports.**

**PROGRAM-7:**

Program for Connection-Oriented Iterative Server in which Server calculates the Net-salary of an Employee based on the following details sent by the Client: BASIC-SAL, HRA, DA, PT and EPF.

**Description:**

An iterative server is a type of server that handles one client connection at a time, sequentially processing requests from clients. In the case of a connection-oriented iterative server, it operates using TCP (Transmission

Control Protocol) instead of UDP (User Datagram Protocol). Unlike UDP, TCP establishes a connection before sending data and guarantees reliable, ordered delivery of packets.

**Algorithm:**

* Import necessary libraries (e.g., socket).
* Define constants for server address and port.
* Create a socket object and bind it to the server address and port.
* Listen for incoming connections.
* Accept incoming connections and get the client socket object and address.
* Inside a loop:
  1. Receive data from the client (BASIC-SAL, HRA, DA, PT, EPF).
  2. Calculate net salary based on the received data.
  3. Send the calculated net salary back to the client.
  4. Close the connection if the client sends a termination signal.
* Close the client socket.
* Close the server socket.

**PROGRAM 7**

**Q)Program for Connection-Oriented Iterative service in which server calculates the Net-salary of an Employee based on the following details sent by the client**

**i)basic-sal**

**ii) hra**

**iii) da**

**iv) pt**

**v) epf ( net-sala=basic+hra+da-pt-epf).**

**Problem Description**: The problem can be implemented using sockets. General implementation steps are as follows:

**Steps involved in writing the Server Process**:

Create a socket using *socket( ) system call.*.

Bind server’s address and port using *bind( ) system call.*

Convert the socket into a listening socket using *listen( ) sytem call.*

Wait for client connection to complete using *accept( ) system call.*

Receive the Client request using recv() system call which consist of the name of the command that is to be executed along with data parameters(if any)

The command is interpreted and executed.

On successful execution the result is passed back to the client by the server

**Steps involved in writing the Client Process**:

Create a socket.

Fill in the internet socket address structure (with server information).

Connect to server using *connect system call*.

The client passes the command and data parameters (if any) to the server.

Read the result sent by the server, write it to standard output.

Close the socket connection.

**Execution Procedure:** Suppose, the server program is server.c and client program is client.c First compile the Server program as,

$ cc server.c – o obj => $ ./obj& => $ cc client.c => $./a.out

**Validation:**

**Sample Input:** The Client sends the salary details 1000 2000 3000 500 500 100 **Sample Output**

Then it will return the complete salaray after calculation 4900

**//Server program**

#include<sys/types.h>

#include<sys/socket.h>

#include<stdio.h>

#include<netinet/in.h>

#include<unistd.h>

#include<arpa/inet.h>

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

{ int i,bp,hra,pt,da,epf,c,consd; //char msg[10]; int sockfd=socket(AF\_INET,SOCK\_STREAM,0); struct sockaddr\_in ser,cli; ser.sin\_family=AF\_INET; ser.sin\_port=htons(atoi(argv[2]));

ser.sin\_addr.s\_addr=inet\_addr(argv[1]);

bind(sockfd,(struct sockaddr \*)&ser,sizeof(ser));

listen(sockfd,5);

for(;;)

{

int len=sizeof(cli);

int consd=accept(sockfd,(struct sockaddr \*)&cli,&len);

recv(consd,(int \*)&bp,sizeof(bp),0);

recv(consd,(int \*)&hra,sizeof(hra),0); recv(consd,(int \*)&da,sizeof(da),0); recv(consd,(int \*)&pt,sizeof(pt),0); recv(consd,(int \*)&epf,sizeof(epf),0); printf("after receving msg to server\n"); printf("Server will send back service to client\n");

c=bp+hra+da-epf-pt; send(consd,(int \*)&c,sizeof(c),0);

}

close(consd);

close(sockfd);

}

**//client program**

#include<sys/types.h>

#include<sys/socket.h>

#include<stdio.h>

#include<netinet/in.h>

#include<unistd.h>

#include<arpa/inet.h>

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

{

int bp,hra,pt,da,epf,c;

// char msg[10],msg1[10]; int sockfd=socket(AF\_INET,SOCK\_STREAM,0); struct sockaddr\_in ser; ser.sin\_family=AF\_INET; ser.sin\_port=htons(atoi(argv[2])); ser.sin\_addr.s\_addr=inet\_addr(argv[1]); connect(sockfd,(struct sockaddr \*)&ser,sizeof(ser));

printf("enter Bp,hra,da,pt,epf \n"); scanf("%d%d%d%d%d",&bp,&hra,&da,&pt,&epf);

send(sockfd,(int \*)&bp,sizeof(bp),0); send(sockfd,(int \*)&hra,sizeof(hra),0);

send(sockfd,(int \*)&da,sizeof(da),0); send(sockfd,(int \*)&pt,sizeof(pt),0); send(sockfd,(int \*)&epf,sizeof(epf),0);

recv(sockfd,(int \*)&c,sizeof(c),0);

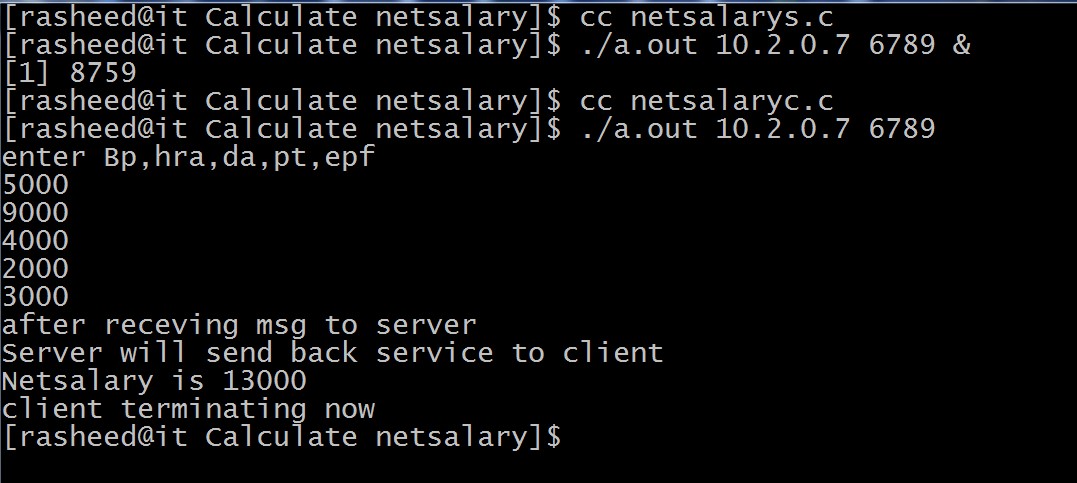
printf("Netsalary is %d\n", c);

printf("client terminating now\n");

close(sockfd);

}

**OUTPUT:**



**PROGRAM-8:**

**Aim:**

Write a program wherein the Client sends the Roll Number and marks of a student to the Server, Server responds to it by sending the division.

Implement the above using Connectionless service.(Service:Student Guide)

**Description:**

In a connectionless service named "Student Guide," a client sends the roll number and marks of a student to the server, which calculates the division based on the marks received. Utilizing protocols like UDP, the server responds to the client with the division information without the need for a persistent connection. This approach allows for quick data exchanges without the overhead of connection management, suitable for scenarios where real-time responses are needed, such as in student information retrieval and division calculation.

**Algorithm:**

* Import necessary libraries (e.g., socket).
* Define constants for server address and port.
* Create a socket object using UDP protocol.
* Define server address and port.
* Receive data (roll number and marks) from the client.
* Calculate the division based on the received marks.
* Send the division back to the client.
* Close the socket.

**PROGRAM 8**

**Q)Write a program wherein the Client sends the Roll Number and marks of a student to the Server, Server responds to it by sending the division. Implement the above using Connectionless service.(serive:Student Grade)**

**//Server program**

#include<sys/types.h>

#include<sys/socket.h>

#include<stdio.h>

#include<netinet/in.h>

#include<unistd.h>

#include<arpa/inet.h> #include<string.h>

main()

{

int sockfd,clilen,i,marks1,marks2,marks3;

int percentage;

char division[60]; struct sockaddr\_in sa,ca; sockfd=socket(AF\_INET,SOCK\_DGRAM,0); sa.sin\_family=AF\_INET; sa.sin\_port=htons(6500); sa.sin\_addr.s\_addr=inet\_addr("10.2.0.7"); bind(sockfd,(struct sockaddr \*)&sa,sizeof(sa));

for(;;) { clilen=sizeof(ca);

recvfrom(sockfd,(int \*)&marks1,20,0,(struct sockaddr \*)&ca,&clilen); recvfrom(sockfd,(int \*)&marks2,20,0,(struct sockaddr \*)&ca,&clilen); recvfrom(sockfd,(int \*)&marks3,20,0,(struct sockaddr \*)&ca,&clilen);

percentage=(marks1+marks2+marks3)/3;

if(percentage>=80)

{

strcpy(division,"DISTINCTION");

}

else if(percentage>=70 && percentage<80)

{

strcpy(division,"FIRST DIVISION");

}

else if(percentage>=60 && percentage<70)

{

strcpy(division,"SECOND DIVISION");

}

else if(percentage>=40 && percentage<60)

{

strcpy(division,"THIRD DIVISION");

}

else

{

strcpy(division,"FAILED");

}

sendto(sockfd,division,60,0,(struct sockaddr \*)&ca,clilen);

} close(sockfd);

}

**//client program**

#include<sys/types.h>

#include<sys/socket.h>

#include<stdio.h>

#include<netinet/in.h>

#include<unistd.h>

#include<arpa/inet.h>

main()

{

int sockfd,servlen,marks1,marks2,marks3,percentage;

struct sockaddr\_in sa; char division[20]; sockfd=socket(AF\_INET,SOCK\_DGRAM,0); sa.sin\_family=AF\_INET; sa.sin\_port=htons(6500); sa.sin\_addr.s\_addr=inet\_addr("10.2.0.7"); servlen=sizeof(sa);

printf("enter the marks of all subjects \n"); scanf("%d",&marks1); printf("second subject \n"); scanf("%d",&marks2); printf("third subject \n"); scanf("%d",&marks3);

sendto(sockfd,(int \*)&marks1,sizeof(marks1),0,(struct sockaddr \*)&sa,servlen); sendto(sockfd,(int \*)&marks2,sizeof(marks2),0,(struct sockaddr \*)&sa,servlen); sendto(sockfd,(int \*)&marks3,sizeof(marks3),0,(struct sockaddr \*)&sa,servlen);

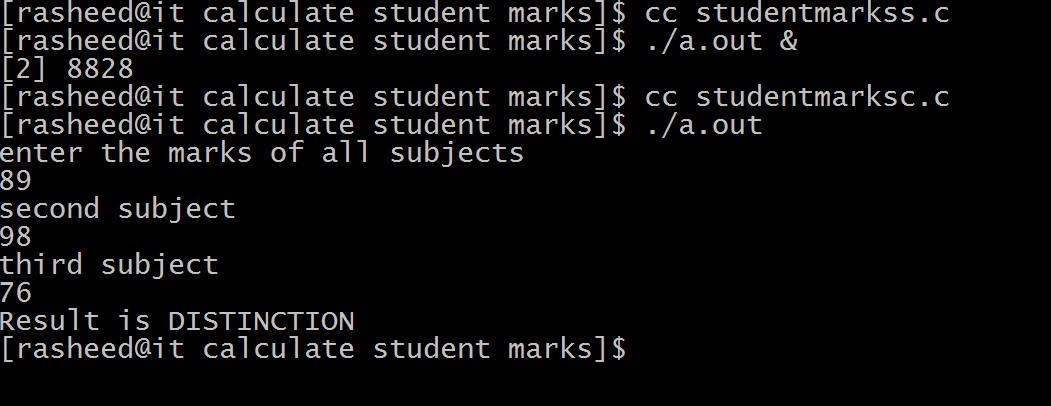
// recvfrom(sockfd,(int \*)&percentage,sizeof(percentage),0,(struct sockaddr \*)&sa,&servlen);

recvfrom(sockfd,division,sizeof(division),0,(struct sockaddr \*)&sa,&servlen); printf("Result is %s \n",division);

close(sockfd);

}

**OUTPUT:**



**PROGRAM-9:**

**Aim:**

Program to demonstrate the usage of Advanced socket system calls like getsockopt(), setsockopt(), getpeername(), readv(), writev() and getsockname().

**Description:**

These advanced socket system calls provide additional control and information retrieval capabilities when working with sockets in networking applications.

1. **getsockopt():**
   * **Purpose:** Used to retrieve options associated with a socket.  **Syntax:** getsockopt(socket, level, option)
   * **Example:**

import socket

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) value = s.getsockopt(socket.IPPROTO\_TCP, socket.TCP\_NODELAY) print("TCP\_NODELAY option value:", value)

1. **setsockopt():**
   * **Purpose:** Used to set options associated with a socket.  **Syntax:** setsockopt(socket, level, option, value)
   * **Example:**

import socket

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.setsockopt(socket.IPPROTO\_TCP, socket.TCP\_NODELAY, 1)

1. **getpeername():**

* **Purpose:** Returns the remote address to which the socket is connected.
* **Syntax:** getpeername(socket)  **Example:**

import socket

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.connect(('www.example.com', 80)) peername = s.getpeername() print("Connected to:", peername)

1. **readv():**
   * **Purpose:** Reads data from multiple buffers and stores it into a single buffer.
   * **Syntax:** readv(socket, buffers)
   * **Example:** (Note: readv is not directly available in Python's socket module. You may need to use recv\_into() or similar methods to achieve similar functionality.)
2. **writev():**
   * **Purpose:** Writes data from multiple buffers into the socket.
   * **Syntax:** writev(socket, buffers)
   * **Example:** (Note: writev is not directly available in Python's socket module. You may need to use send() or similar methods to achieve similar functionality.)
3. **getsockname():**
   * **Purpose:** Returns the socket's own address.  **Syntax:** getsockname(socket)  **Example:**

import socket

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.bind(('localhost', 12345)) sockname = s.getsockname() print("Socket bound to:", sockname)

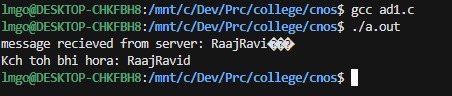
**PROGRAM 9**

**Q)Programs to demonstrate the usage of Advanced socket system calls like getsockopt( ) setsockopt( ), getpeername ( ), readv( ), writev( ), getsockname( ) .**

1. **readv, writev**

#include <stdio.h>#include <sys/types.h>#include <sys/socket.h>#include <arpa/inet.h>#include <sys/uio.h>// program to demonstrate readv and writev int main() { int sockfd = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP); struct iovec iov1[2], iov2[2]; struct sockaddr\_in sa; sa.sin\_family = AF\_INET; sa.sin\_port = htons(6005); sa.sin\_addr.s\_addr = inet\_addr("127.0.0.1"); connect(sockfd, (struct sockaddr\*)&sa, sizeof(sa)); char msg1[20] = "Raaj", msg2[20] = "Ravi", msg3[20]; iov1[0].iov\_base = &msg1; iov1[0].iov\_len = 4; iov1[1].iov\_base = &msg2; iov1[1].iov\_len = 4; iov2[0].iov\_base = &msg3; iov2[0].iov\_len = 20; writev(sockfd, iov1, 2); readv(sockfd, iov2, 2); printf("Message received: %s\n", msg3); close(sockfd); }

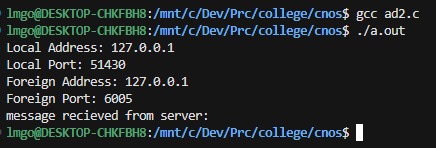
**Output:**

****

**ii) getsockname, getpeername**

#include <stdio.h>#include <sys/types.h>#include <sys/socket.h>#include <arpa/inet.h>#include <sys/uio.h>#include <string.h>#include <unistd.h>#include <stdlib.h>// program to demonstrate getsockname and setsockname int main() { int sockfd = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP); struct sockaddr\_in sa, local; sa.sin\_family = AF\_INET; sa.sin\_port = htons(6005); sa.sin\_addr.s\_addr = inet\_addr("127.0.0.1"); connect(sockfd, (struct sockaddr\*)&sa, sizeof(sa)); int sa\_size = sizeof(sa); getsockname(sockfd, (struct sockaddr\*)&local, &sa\_size); printf("Local Address: %s\n", inet\_ntoa(local.sin\_addr)); printf("Local Port: %d\n", ntohs(local.sin\_port)); getpeername(sockfd, (struct sockaddr\*)&local, &sa\_size); printf("Foreign Address: %s\n", inet\_ntoa(local.sin\_addr)); printf("Foreign Port: %d\n", ntohs(local.sin\_port)); close(sockfd);}

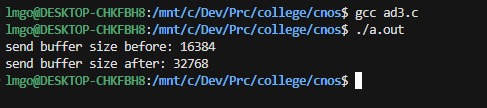
**Output:**

****

**iii) getsockopt, setsockopt**

#include <stdio.h>#include <netinet/in.h>#include <netinet/tcp.h>#include <sys/socket.h>// program to demonstrate getsockopt and setsockoptint main() { int sockfd = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP); int sendbuff; int optlen = sizeof(sendbuff); getsockopt(sockfd, SOL\_SOCKET, SO\_SNDBUF, (char\*)&sendbuff, &optlen); printf("send buffer size before: %d\n", sendbuff); sendbuff = 16384; setsockopt(sockfd, SOL\_SOCKET, SO\_SNDBUF, (char\*)&sendbuff, sizeof(sendbuff)); getsockopt(sockfd, SOL\_SOCKET, SO\_SNDBUF, (char\*)&sendbuff, &optlen); printf("send buffer size after: %d\n", sendbuff);}

**Output:**

****