| Ex. No. 1a | BASIC LINUX COMMANDS | Date : |
|---|---|---|

**a) Basics**

1. *echo* SRM → to display the string SRM
2. *clear* → to clear the screen
3. *date* → to display the current date and time
4. *cal* 2003 → to display the calendar for the year 2003
   *cal* 6 2003 → to display the calendar for the June-2003
5. *passwd* → to change password

**b) Working with Files**

1. *ls* → list files in the present working directory
   *ls –l* → list files with detailed information (long list)
   *ls –a* → list all files including the hidden files
2. cat > f1 → to create a file (Press ^d to finish typing)
3. *cat* f1 → display the content of the file f1
4. *wc* f1 → list no. of characters, words & lines of a file f1
   *wc –c* f1 → list only no. of characters of file f1
   *wc –w* f1 → list only no. of words of file f1
   *wc –l* f1 → list only no. of lines of file f1
5. *cp* f1 f2 → copy file f1 into f2
6. *mv* f1 f2 → rename file f1 as f2
7. *rm* f1 → remove the file f1
8. *head* –5 f1 → list first 5 lines of the file f1
   *tail* –5 f1 → list last 5 lines of the file f1

**c) Working with Directories**

1. *mkdir* elias → to create the directory elias
2. *cd* elias → to change the directory as elias
3. *rmdir* elias → to remove the directory elias
4. *pwd* → to display the path of the present working directory
5. *cd* → to go to the home directory
   *cd ..* → to go to the parent directory
   *cd -* → to go to the previous working directory
   *cd /* → to go to the root directory

**d) File name substitution**

1. *ls f?* → list files start with 'f' and followed by any one character
2. *ls* *.c → list files with extension 'c'
3. *ls* [gpy]et → list files whose first letter is any one of the character g, p or y and followed by the word et
4. *ls* [a-d,l-m]ring → list files whose first letter is any one of the character from a to d and l to m and followed by the word ring.

**e) I/O Redirection**

1. Input redirection

   *wc –l* < ex1 → To find the number of lines of the file 'ex1'

2. Output redirection

   *who* > f2 → the output of 'who' will be redirected to file f2

3. *cat* >> f1 → to append more into the file f1

**f) Piping**

Syntax : Command1 | command2

Output of the command1 is transferred to the command2 as input. Finally output of the command2 will be displayed on the monitor.

ex. *cat* f1 | *more* → list the contents of file f1 screen by screen

*head* –6 f1 |*tail* –2 → prints the $5^{th}$ & $6^{th}$ lines of the file f1.

**g) Environment variables**

1. *echo $HOME* → display the path of the home directory
2. *echo $PS1* → display the prompt string $
3. *echo $PS2* → display the second prompt string ( > symbol by default )
4. *echo $LOGNAME* → login name
5. *echo $PATH* → list of pathname where the OS searches for an executable file

**h) File Permission**

-- chmod command is used to change the access permission of a file.

*Method-1*

Syntax : *chmod* [ugo] [+/-] [ rwxa ] filename

u : user, g : group, o : others
+ : Add permission - : Remove the permission
r : read, w : write, x : execute, a : all permissions

ex. *chmod* ug+rw f1
adding 'read & write' permissions of file f1 to both user and group members.

*Method-2*

Syntax : *chmod* octnum file1

The 3 digit octal number represents as follows

- first digit -- file permissions for the user
- second digit -- file permissions for the group
- third digit -- file permissions for others

Each digit is specified as the sum of following
4 – read permission, 2 – write permission, 1 – execute permission

ex. *chmod* 754 f1

it change the file permission for the file as follows

- read, write & execute permissions for the user ie; 4+2+1 = 7
- read, & execute permissions for the group members ie; 4+0+1 = 5
- only read permission for others ie; 4+0+0 = 4

| Ex. No. 1b | FILTERS and ADMIN COMMANDS | Date : |
|---|---|---|

**FILTERS**

**1. cut**

- Used to cut characters or fileds from a file/input

  Syntax : **cut** **-c**chars filename
  **-f**fieldnos filename

- By default, tab is the filed separator(delimiter). If the fileds of the files are separated by any other character, we need to specify explicitly by **–d** option

  **cut** **-d**ddelimitchar **-f**fileds filname

**2. grep**

- Used to search one or more files for a particular pattern.

  Syntax : **grep** pattern filename(s)

  - Lines that contain the *pattern* in the file(s) get displayed
  - pattern can be any regular expressions
  - More than one files can be searched for a pattern

**-v** option displays the lines that do not contain the *pattern*
**-l** list only name of the files that contain the *pattern*
**-n** displays also the line number along with the lines that matches the *pattern*

**3. sort**

- Used to sort the file in order

  Syntax : **sort** filename

  - Sorts the data as text by default
  - Sorts by the first filed by default

**-r** option sorts the file in descending order
**-u** eliminates duplicate lines
**-o** filename writes sorted data into the file *fname*
**-t**dchar sorts the file in which fileds are separated by *dchar*
**-n** sorts the data as number
**+1n** skip first filed and sort the file by second filed numerically

**4. Uniq**

- Displays unique lines of a sorted file

  Syntax : **uniq** filename

**-d** option displays only the duplicate lines
**-c** displays unique lines with no. of occurrences.

**5. diff**

- Used to differentiate two files

Syntax : **diff** f1 f2
compare two files f1 & f2 and prints all the lines that are differed between f1 & f2.

**Q1.** Write a command to cut 5 to 8 characters of the file *f1*.
$

**Q2.** Write a command to display user-id of all the users in your system.
$

**Q3.** Write a command to check whether the user j*udith* is available in your system or not. (use grep)
$

**Q4.** Write a command to display the lines of the file *f1* starts with SRM.
$

**Q5.** Write a command to sort the file /etc/passwd in descending order
$

**Q6.** Write a command to display the unique lines of the sorted file *f21*. Also display the number of occurrences of each line.
$

**Q7.** Write a command to display the lines that are common to the files *f1* and *f2*.
$

## EX:NO:2 Process Creation using fork() and Usage of getpid(), getppid(), wait() functions

**Aim :**

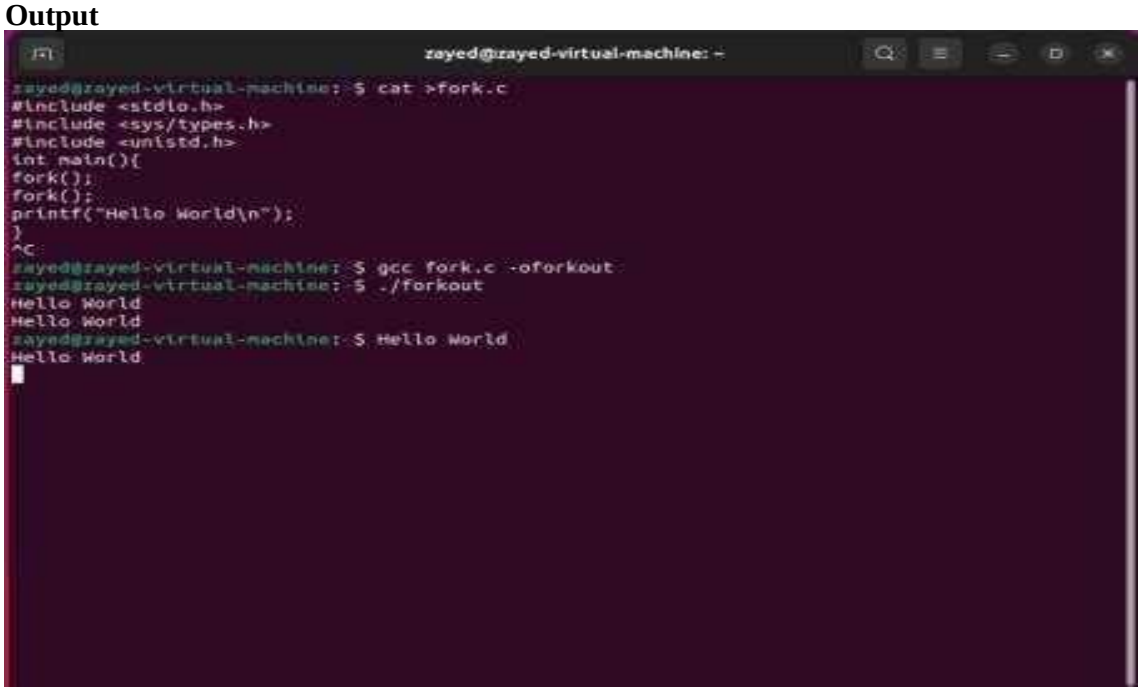
To write a program for process Creation using fork() and usage of getpid(), getppid(), wait() function.

**Program :**

- Process creating using fork()

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int
main(){
fork();
fork();
printf("Hello World\n");
}
```

**Output**




- Usage of getpid() and getppid()

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void main()
{
```

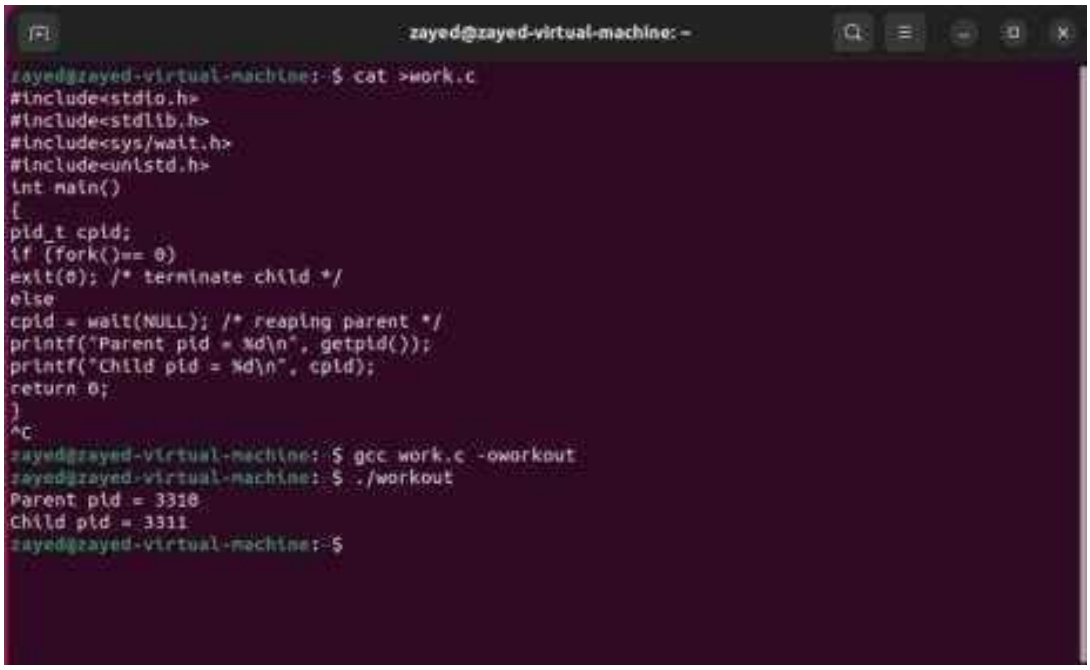
```
//variable to store calling function's
int process_id, p_ process_id;
//getpid() - will return process id of calling function
process_id = getpid();
//getppid() - will return process id of parent function
p_process_id = getppid();
//printing the process ids
printf("The process id: %d\n",process_id);
printf("The process id of parent function: %d\n",p_process_id);
 }
```

**Output :**


- Usage of wait()

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
pid_t cpid;
if (fork()== 0)
exit(0); /* terminate child */
else
cpid = wait(NULL); /* reaping parent */
printf("Parent pid = %d\n", getpid());
printf("Child pid = %d\n", cpid);
return 0;
 }
```

**Output:**




**Result :**

Thus Successfully completed Process Creation using fork() and Usage of getpid(), getppid(), wait() functions.

# EX:NO 3 Multithreading and pthread in C

**Aim :**

To implement and study Multithreading and pthread in C

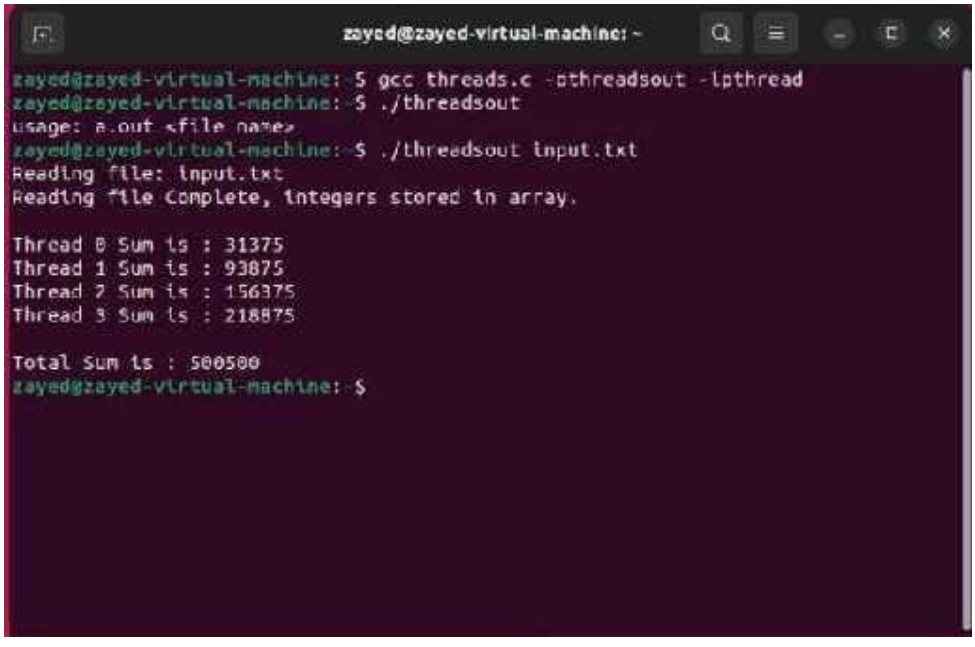
**Program :**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX 1000
#define MAX_THREAD 4

int array[1000];
int sum[4] = { 0 };
int arraypart = 0;

void* sum_array(void* arg)
{
    int thread_part = arraypart++;

  for (int i = thread_part * (MAX / 4); i < (thread_part + 1) * (MAX / 4); i++)
  {
    sum[thread_part] += array[i];
  }
}

void testSum()
{
 pthread_t threads[MAX_THREAD];

  for (int i = 0; i < MAX_THREAD; i++)
  {
    pthread_create(&threads[i], NULL, sum_array, (void*)NULL);
  }

  // joining threads
  for (int i = 0; i < MAX_THREAD; i++)
  {
    pthread_join(threads[i], NULL);
  }

   // print each thread
  for (int i = 0; i < MAX_THREAD; i++)
  {
    printf("Thread %d Sum is : %d \n",i, sum[i]);
  }
```

```
    // adding the 4 parts
    int total_sum = 0;
    for (int i = 0; i < MAX_THREAD; i++)
    {
        total_sum += sum[i];
    }
      printf("\nTotal Sum is : %d \n",total_sum);

}
void readfile(char* file_name)
{
  char ch;

  FILE *fp;


  fp = fopen(file_name,"r"); // read mode

  if( fp == NULL )
  {
    perror("Error while opening the file.\n");
    exit(EXIT_FAILURE);
  }


   char line [5]; /* line size */

   int i=0;

   printf("Reading  file:  ");
   fputs(file_name,stdout);
   printf("\n");


   while ( fgets ( line, sizeof line, fp) != NULL ) /* read a line */
   {
     if (i < 1000)
     {
       array[i]=atoi(line);
     }

    i++;
   }
```

```
  fclose(fp);

   printf("Reading file Complete, integers stored in array.\n\n");

}


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

  if (argc != 2) {
        fprintf(stderr,"usage: a.out <file name>\n");
        /*exit(1);*/
        return -1;
   }

  readfile(argv[1]);


  //Debug code for testing only
  testSum();

  return 0;
}
```

**Output :**




**Result :**

Successfully implemented and studied Multithreading and pthread in C

# EX:NO 4 Mutual Exclusion using semaphore and monitor

**Aim :**

To implement Mutual Exclusion using semaphore and monitor

**Program :**

**USING SEMAPHORE**

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>
#include<errno.h>
#include <stdlib.h>
#include<sched.h>
int philNo[5] = { 0, 1, 2, 3, 4 };
// my_semaphore structure
typedef struct {

  // Semaphore mutual exclusion variable
  pthread_mutex_t mutex;
  // Semaphore count variable
  int cnt;
  // Semaphore conditonal variable
  pthread_cond_t conditional_variable;
}
 my_semaphore;

// Function to initialise the semaphore variables
int init(my_semaphore *sema, int pshared, int val) {
  // The case when pshared == 1 is not implemeted as it was not required because the
philosphers are implemented using threads and not processes.
```

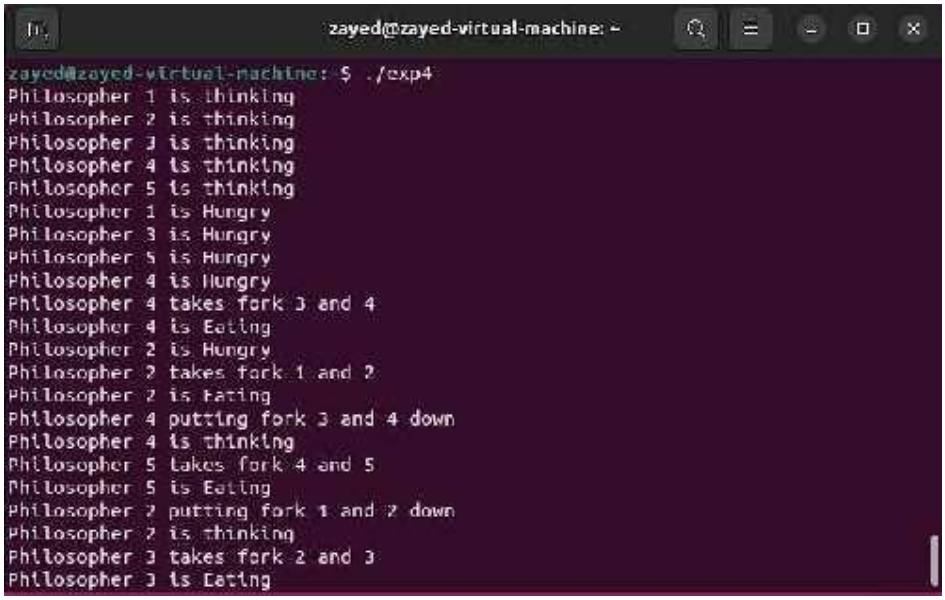
```
 if(pshared == 1){
        printf("Cannot handle semaphores shared between processes!!! Exiting\n");
        return -1;
 }

 // Initialisng the semaphore conditonal variable
 pthread_cond_init(&sema->conditional_variable, NULL);

 // Initialisng the semaphore count variable
 sema->cnt = val;

 // Initialisng the semaphore mutual exclusion variable
 pthread_mutex_init(&sema->mutex, NULL);

 return 0;
}

int signal(my_semaphore *sema) {

 //This locks the mutex so that only thread can access the critical section at a time
 pthread_mutex_lock(&sema->mutex);
 sema->cnt = sema->cnt + 1;

 // This wakes up one waiting thread
 if (sema->cnt)
        pthread_cond_signal(&sema->conditional_variable);
```

```
  // A woken thread must acquire the lock, so it will also have to wait until we call unlock

  // This releases the mutex
  pthread_mutex_unlock(&sema->mutex);

  return 0;
}
int wait(my_semaphore *sema) {
  //This locks the mutex so that only thread can access the critical section at a time
  pthread_mutex_lock(&sema->mutex);
  // While the semaphore count variable value is 0 the mutex is blocked on the conditon
variable
  while (!(sema->cnt))
    pthread_cond_wait(&sema->conditional_variable, &sema->mutex);
    // unlock mutex, wait, relock mutex

  sema->cnt = sema->cnt - 1;

  // This releases the mutex and threads can access mutex
  pthread_mutex_unlock(&sema->mutex);
  return 0;
}
// Print semaphore value for debugging
void signal1(my_semaphore *sema) {
  printf("Semaphore variable value = %d\n", sema->cnt);
}
// Declaring the semaphore variables which are the shared resources by the threads
my_semaphore forks[5], bowls;

//Function for the philospher threads to eat
void *eat_food(void *arg) {
    while(1) {
        int* i = arg;
        // This puts a wait condition on the bowls to be used by the current philospher so
that the philospher can access these forks whenever they are free
        wait(&bowls);
```

```
        // This puts a wait condition on the forks to be used by the current philospher so
that the philospher can access these forks whenever they are free
        wait(&forks[*i]);
        wait(&forks[(*i+4)%5]);
        sleep(1);
      //Print the philospher number, its thread ID and the number of the forks it uses for
        eating printf("Philosopher %d with ID %ld eats using forks %d and %d\n", *i+1,
pthread_self(), *i+1, (*i+4)%5+1);
       // This signals the other philospher threads that the bowls are    available for
eating signal(&bowls);
        // This signals the other philospher threads that these forks are available for
eating and thus other threads are woken up
        signal(&forks[*i]);
        signal(&forks[(*i+4)%5]);
        sched_yield();
    }
}
void main() {
    int i = 0;
    // Initialising the forks (shared variable) semaphores
    while(i < 5){
        init(&forks[i], 0, 1);
        i++;
    }
    // Initialising the bowl (shared variable) semaphore
    init(&bowls, 0, 1);
    // Declaring the philospher threads
    pthread_t phil[5];
    i = 0;
    // Creating the philospher threads
    while(i < 5) {
        pthread_create(&phil[i], NULL, eat_food, &philNo[i]);
```

```
    i++;
  }
  i = 0;
  // Waits for all the threads to end their execution before ending
  while(i < 5) {
    pthread_join(phil[i], NULL);
    i++;
  }}
```

**Output :**



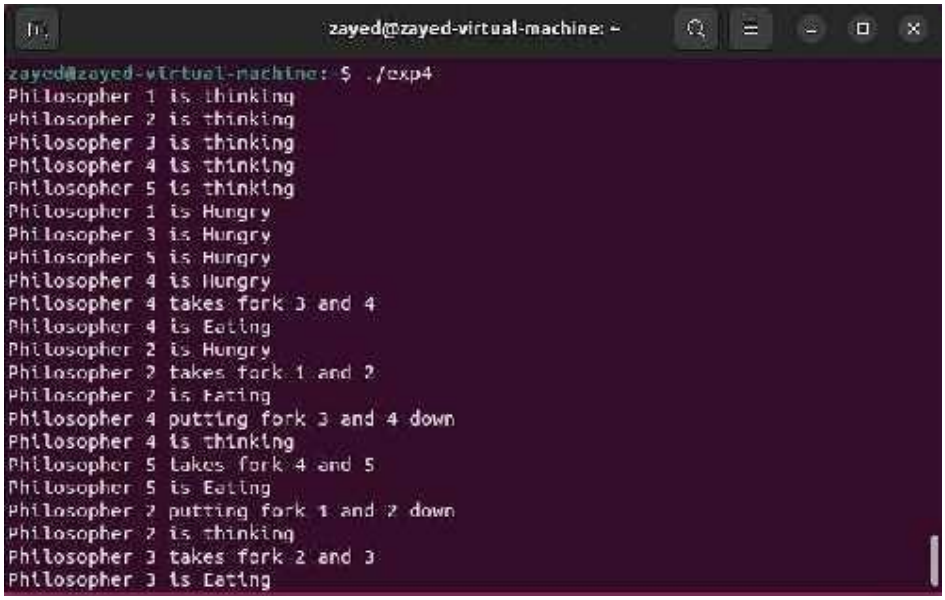

**USING MONITOR**

```
monitor DP
{
   status state[5];
condition self[5];
// Pickup chopsticks
Pickup(int i)
{
   // indicate that I'm hungry
   state[i] = hungry;
   // set state to eating in test()
   // only if my left and right neighbors
   // are not eating
   test(i);
```

```
    // if unable to eat, wait to be signaled if
    (state[i] != eating)
        self[i].wait;
}
// Put down chopsticks
Putdown(int i)
{
    // indicate that I'm thinking
    state[i] = thinking;
    // if right neighbor R=(i+1)%5 is hungry and
    // both of R's neighbors are not eating,
    // set R's state to eating and wake it up by
    // signaling R's CV
        test((i + 1) % 5);
        test((i + 4) % 5);
    }
    test(int i)
    {
        if (state[(i + 1) % 5] != eating
            && state[(i + 4) % 5] != eating
            && state[i] == hungry) {
            // indicate that I'm eating
            state[i] = eating;
            // signal() has no effect during Pickup(),
            // but is important to wake up waiting
            // hungry philosophers during Putdown()
            self[i].signal();
        }
    }
    init()
    {
        // Execution of Pickup(), Putdown() and test()
        // are all mutually exclusive,
        // i.e. only one at a time can be executing
for
    i = 0 to 4
```

```
        // Verify that this monitor-based solution is
        // deadlock free and mutually exclusive in that
        // no 2 neighbors can eat simultaneously
        state[i] = thinking;
    }
}
```

**Output :**




**Result :**

Successfully executed Mutual Exclusion using semaphore and monitor

# EX:NO:5 Reader-Writer problem

**Aim** :

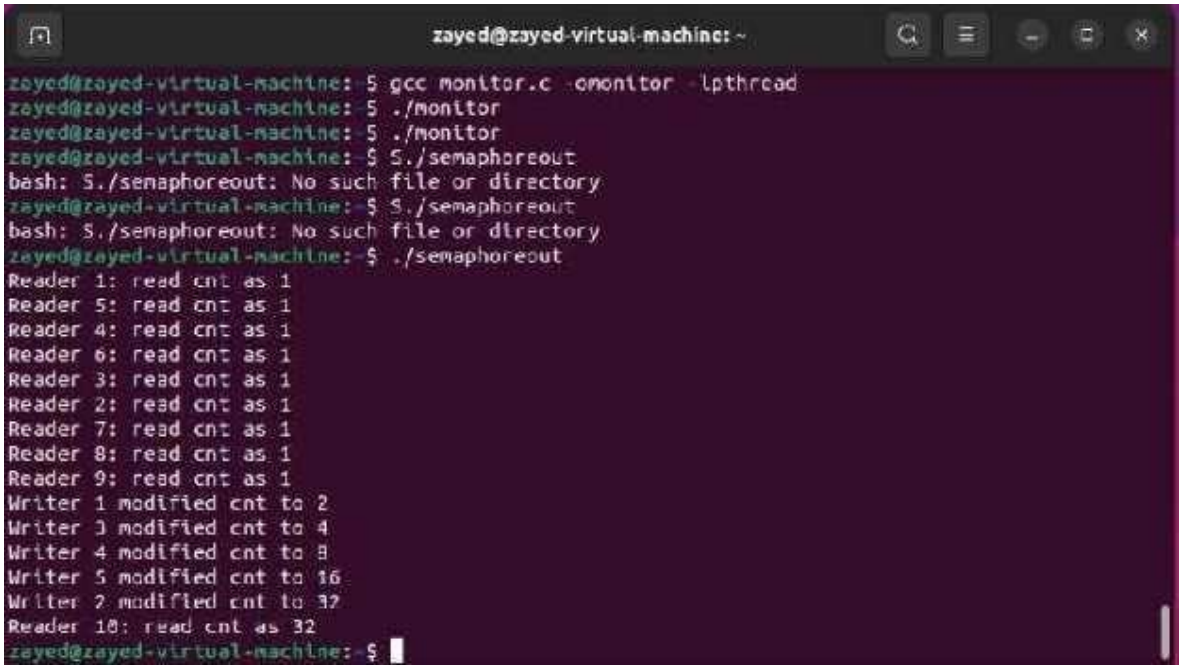
To study the Reader – Writer problem

**Program :**

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
sem_t wrt;
pthread_mutex_t
mutex; int cnt = 1;
int numreader = 0;
void *writer(void *wno)
{
   sem_wait(&wrt);
   cnt = cnt*2;
   printf("Writer %d modified cnt to %d\n",(*((int *)wno)),cnt);
   sem_post(&wrt);
}
void *reader(void *rno)
{
   // Reader acquire the lock before modifying numreader
   pthread_mutex_lock(&mutex);
   numreader++;
   if(numreader == 1) {
      sem_wait(&wrt); // If this id the first reader, then it will block the writer
   }
   pthread_mutex_unlock(&mutex);
   // Reading Section
   printf("Reader %d: read cnt as %d\n",*((int *)rno),cnt);
```

```
    // Reader acquire the lock before modifying numreader
    pthread_mutex_lock(&mutex);
    numreader--;
    if(numreader == 0) {
        sem_post(&wrt); // If this is the last reader, it will wake up the writer.
    }
    pthread_mutex_unlock(&mutex);
}
int main()
{
    pthread_t read[10],write[5];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&wrt,0,1);
    int a[10] = {1,2,3,4,5,6,7,8,9,10}; //Just used for numbering the producer and consumer
    for(int i = 0; i < 10; i++) {
        pthread_create(&read[i], NULL, (void *)reader, (void *)&a[i]);
    }
    for(int i = 0; i < 5; i++) {
        pthread_create(&write[i], NULL, (void *)writer, (void *)&a[i]);
    }
    for(int i = 0; i < 10; i++) {
        pthread_join(read[i], NULL);
    }
    for(int i = 0; i < 5; i++) {
        pthread_join(write[i], NULL);
    }
    pthread_mutex_destroy(&mutex);
    sem_destroy(&wrt);
    return 0;

}
```

**Output:**




**Result:**

Thus Successfully provided a solution to Reader – Writer using mutex and semaphore.

# EX:NO:6 Dining Philosopher's Problem

**Aim:**

To implement and study Dining Philosopher's Problem

**Program:**

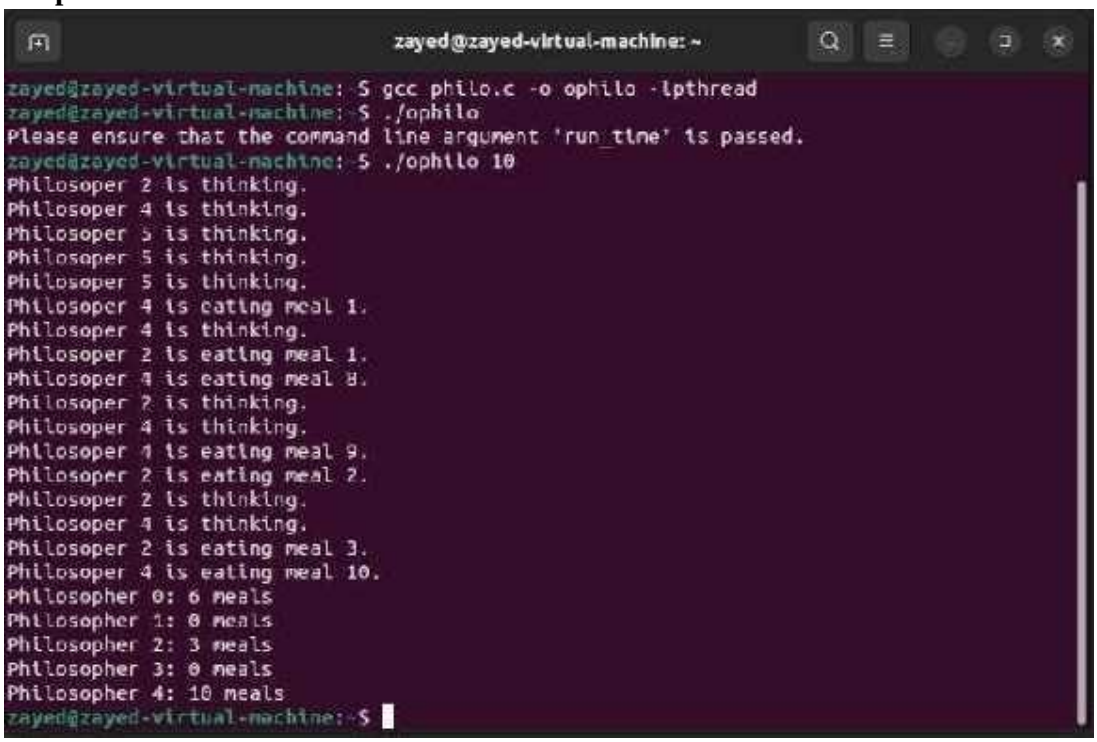
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

//Function declarations
void *pickup_forks(void * philosopher_number); void *return_forks(void *
philosopher_number); void test(int philosopher_number);
int left_neighbor(int philosopher_number); int right_neighbor(int
philosopher_number); double think_eat_time(void);
void think(double think_time);
void eat(double eat_time);

//Constants to be used in the program.
#define PHILOSOPHER_NUM 5
#define MAX_MEALS 10
#define MAX_THINK_EAT_SEC 3

//States of philosophers.
enum {THINKING, HUNGRY, EATING} state[PHILOSOPHER_NUM];

//Array to hold the thread identifiers.
pthread_t philos_thread_ids[PHILOSOPHER_NUM];

//Mutex lock.
pthread_mutex_t mutex;

//Condition variables.
pthread_cond_t cond_vars[PHILOSOPHER_NUM];

//Array to hold the number of meals eaten for each philosopher.
int meals_eaten[PHILOSOPHER_NUM];

int main(int argc, char *argv[])
{
   //Ensure correct number of command line arguments.
   if(argc != 2)
```

```
    {
        printf("Please ensure that the command line argument 'run_time' is passed.\n");
    }
    else
    {
        //Set command line argument value to variable run_time;
        double run_time = atof(argv[1]);

        //Initialize arrays.
        int i;
        for(i = 0; i < PHILOSOPHER_NUM; i++)
        {
            state[i] = THINKING;
            pthread_cond_init(&cond_vars[i], NULL);
            meals_eaten[i] = 0;
        }
        //Initialize the mutex lock.
        pthread_mutex_init(&mutex, NULL);
        //Join the threads.
        for(i = 0; i < PHILOSOPHER_NUM; i++)
        {
            pthread_join(philos_thread_ids[i], NULL);
        }
        //Create threads for the philosophers.
        for(i = 0; i < PHILOSOPHER_NUM; i++)
        {
            pthread_create(&philos_thread_ids[i], NULL, pickup_forks, (void *)&i);
        }
        sleep(run_time);
        for(i = 0; i < PHILOSOPHER_NUM; i++)
        {
            pthread_cancel(philos_thread_ids[i]);
        }
        //Print the number of meals that each philosopher ate.
        for(i = 0; i < PHILOSOPHER_NUM; i++)
        {
            printf("Philosopher %d: %d meals\n", i, meals_eaten[i]);
        }
    }
    return 0;
}
void *pickup_forks(void * philosopher_number)
{
    int loop_iterations = 0;
    int pnum = *(int *)philosopher_number;
    while(meals_eaten[pnum] < MAX_MEALS)
```

```
    {
        printf("Philosoper %d is thinking.\n", pnum);
        think(think_eat_time());

        pthread_mutex_lock(&mutex);
        state[pnum] = HUNGRY;
        test(pnum);

        while(state[pnum] != EATING)
        {
            pthread_cond_wait(&cond_vars[pnum], &mutex);
        }
        pthread_mutex_unlock(&mutex);


        (meals_eaten[pnum])++;


        printf("Philosoper %d is eating meal %d.\n", pnum, meals_eaten[pnum]);

        eat(think_eat_time());

        return_forks((philosopher_number));

        loop_iterations++;
    }
}
void *return_forks(void * philosopher_number)
{
    pthread_mutex_lock(&mutex);
    int pnum = *(int *)philosopher_number;
    state[pnum] = THINKING;

    test(left_neighbor(pnum));
    test(right_neighbor(pnum));
    pthread_mutex_unlock(&mutex);
}
int left_neighbor(int philosopher_number)
{
    return ((philosopher_number + (PHILOSOPHER_NUM - 1)) % 5);
}
int right_neighbor(int philosopher_number)
{
    return ((philosopher_number + 1) % 5);
}
void test(int philosopher_number)
{
    if((state[left_neighbor(philosopher_number)] != EATING) &&
        (state[philosopher_number] == HUNGRY) &&
        (state[right_neighbor(philosopher_number)] != EATING))
    {
        state[philosopher_number] = EATING;
        pthread_cond_signal(&cond_vars[philosopher_number]);
    }
}
```

```
double think_eat_time(void)
{
   return ((double)rand() * (MAX_THINK_EAT_SEC - 1)) / (double)RAND_MAX + 1;
}
void think(double think_time)
{
   sleep(think_time);
}

void eat(double eat_time)
{
   sleep(eat_time);
}
```

**Output:**




**Result:**

Thus Successfully implemented the concepts of Dining Philosophers Problem.