Experiment 3

Familiarization and implementation of programs related to Process and thread.

Process

A process is an active entity which has a text section which contains the program code, data section which contains global variables, a heap that is dynamically allocated during process run time, and a process stack which contains temporary data. A process is created when an executable program is loaded into memory. Two or more processes can be associated with the same program and even though they may share the same text section, they have different stack, data and heap sections.

Threads

A thread is a basic unit of CPU utilization. It has an ID, register set and a stack. Normally a process has a single thread of control. However, processes can have multiple threads of control which will allow them to perform more than one task at a time. Threads within a process share the same code section, data section and operating system resources like open files and signals. Since process creation is time consuming and resource intensive, it is often advantageous to use one process that contains multiple threads to do the same task.

Thread libraries provide users with an API for creating and managing threads. A thread library may be created at the user level or at the kernel level. In the former case, invoking a function in the library results in a local function call, while in the latter case such an invocation will be a system call. One important thread library is the POSIX Pthreads. POSIX standard gives a specification for thread creation and its behaviour. Operating systems, mostly UNIX type systems implement this standard in different ways.

Some Pthread APIs are

1. *pthread_init_attr(&attr)*
   This function sets the attributes of the thread like stack size, scheduling information etc.

2. *pthread_create(&tid, &attr, func, arg)*
   This function is used for creating a thread. It takes the following parameters
i) *tid* - thread ID
ii) *attr* – thread attributes
iii) *func* – the new function where thread execution begins
iv) *arg* – parameter that is passed into the function

3. *pthread_join(tid, NULL)*

The parent thread waits till the child threads terminate and join with the parent. Afterwards, the parent continues execution.

Example:

This program shows how a child process is created to do a task. The program in *expt2.c* creates a child process for finding the sum of two numbers.

expt2.c

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

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

  pid = fork();

  if(pid == 0) // child process
  {
     execv("/home/anil/NetworkLab/EXPT3/prog2", argv);
  }

  wait(NULL);
    exit(0);

}
```

prog2.c

```
#include<stdio.h>

void main ( int argc, char * argv[])
{
  int a,b, sum;

   a = atoi(argv[1]);
   b = atoi(argv[2]);
   sum = a + b;
   printf("sum = %d\n", sum);
}
```

How to get output ?

Find the sum of 3 and 4,

1. Compile *prog2.c* using the command
   gcc -o prog2 prog2.c
2. Compile *expt2.c* using the command
   gcc -o expt2 expt2.c
3. *./expt2* 3 4

will give 7 as the sum. The path for *execv* should be substituted with your path.

After the fork system call, the child process executes the *prog2* program. The parent process waits for the termination of the child process.

The same program can be done using threads
expt3.c

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

void * func_sum(void*);
int sum ;

main(int argc, char * argv[])
{
   pthread_t tid;
   int val[2];
   val[0] = atoi(argv[1]);
   val[1] = atoi(argv[2]);
  pthread_create(&tid,NULL,func_sum,val);
  pthread_join(tid,NULL);
  printf(" sum = %d\n", sum);
}
```

The command for compiling the program is

gcc -o expt3 expt3.c -lpthread

Run the command

./expt3 3 4

Experiment 4

Implementation of first readers writers problem

Description

Readers writers problem is a synchronization problem. There are two types of processes:- writers and readers which are sharing a common resource. A reader process can share the process with other readers but not writers. A writer process requires exclusive access to the resource. A very good example is a file being shared among a set of processes. As long as a reader holds the resource and there are new readers arriving, any writer must wait for the resource to become available.

The first reader accessing the resource must compete with any writers, but once a writer succeeds the other readers can pass directly into the critical section provided that at least one reader is still in the critical section. The *readCount* variable gives the number of readers in the critical section. Only when the last reader leaves the critical section can the writers enter the critical section one at a time which will based on *writeBlock* variable.

Algorithm

The algorithm for the readers writers problem is given below (Gary Nutt)

```
reader() {
  while(TRUE) {
    <other computing>
    P(mutex);
      readCount = readCount +1;
      if(readCount == 1)
        P(writeBlock);
    V(mutex);

/* critical section */
access resource

   P(mutex);
     readCount = readCount – 1;
     if(readCount == 0)
        V(writeBlock);
   V(mutex);
  }
}

writer() {
  while(TRUE) {
    <other computing>;
     P(writeBlock);

/* critical section */
 access resource
```

```
    V(writeBlock);
  }
}
```

The above algorithm is implemented using threads. *Mutex* and *writeBlock* are two semaphores. The first semaphore guards the *readCount* variable while the latter protects the critical section where the resource is shared.

Program

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


void * reader (void *);
void * writer (void *);

sem_t mutex;
sem_t writeBlock;

int readCount =0;

main(int argc, char * argv[])
{
  int i, j, k;

  int N_readers, N_writers;
  int readers_num[100], writers_num[100];

  pthread_t tid_readers[100], tid_writers[100];

  printf("Enter the number of readers:");
  scanf("%d", &N_readers);

  printf("Enter the number of writers:");
  scanf("%d", &N_writers);


  for(k =0; k < N_readers; k++)
  readers_num[k] = k;

  for(k =0; k < N_writers; k++)
```

```
  writers_num[k] = k;


if(sem_init(&mutex, 0, 1) < 0)
 {
    perror("Could not init semaphore mutex");
    exit(1);
 }


if(sem_init(&writeBlock, 0, 1) < 0)
 {
    perror("Could not init semaphore writeBlock");
    exit(1);
 }

for( i =0; i < N_readers; i++)
 {
   if(pthread_create(&tid_readers[i], NULL, reader,  &readers_num[i]  ))
   {
      perror("could not create reader thread");
      exit(1);
   }

 }

for( j=0; j < N_writers; j++)
 {
 if(pthread_create(&tid_writers[j], NULL, writer, &writers_num[j]))
 {
  perror("could not create writer thread");
  exit(1);
 }
 }

 for (i =0; i < N_readers; i++)
 {
   pthread_join(tid_readers[i], NULL);
 }

 for ( j=0; j < N_writers; j++)
 {
  pthread_join(tid_writers[j], NULL);
 }

 sem_destroy(&mutex);
```

```
  sem_destroy(&mutex);

}



void * reader (void* param)
{

    int i = *((int *) param);

     while(1)
      {
          sleep(1);
        if(sem_wait(&mutex) < 0)
         {
            perror("cannot decrement the semaphore mutex");
            exit(1);
         }

         readCount = readCount + 1;
         if(readCount == 1)
          {
             if(sem_wait(&writeBlock) < 0)
              {
                 perror("cannot decrement the semaphore writeBlock");
                 exit(1);
              }
          }

              if( sem_post(&mutex) < 0)
              {
                   perror("cannot increment semaphore mutex");
                   exit(1);
              }

            // READ RESOURCES
              printf("READER %d is READING \n", i);
                sleep(1);



        if(sem_wait(&mutex) < 0)
         {
            perror("cannot decrement the semaphore mutex");
            exit(1);
```

```
        }

        readCount = readCount - 1;
        if(readCount == 0)
         {

            if( sem_post(&writeBlock) < 0)
            {
                perror("cannot increment semaphore mutex");
                exit(1);
            }
         }


            if( sem_post(&mutex) < 0)
            {
                perror("cannot increment semaphore mutex");
                exit(1);
            }

         }


}


void * writer (void * param)
{

   int i = *((int *) param);



   while(1)
    {
        sleep(1);
            if(sem_wait(&writeBlock) < 0)
             {
               perror("cannot decrement the semaphore writeBlock");
               exit(1);
             }

             // WRITE RESOURCES
            printf("WRITER %d IS WRITING \n", i);
            if( sem_post(&writeBlock) < 0)
            {
```

```
                perror("cannot increment semaphore writeBlock");
                exit(1);
            }

    }

}
```

Output

```
anil@anil-300E4Z-300E5Z-300E7Z:~/NetworkLab$ ./expt4

Enter the number of readers:5
Enter the number of writers:3

READER 0 is READING
READER 1 is READING
READER 2 is READING
READER 4 is READING
READER 3 is READING
WRITER 0 IS WRITING
WRITER 1 IS WRITING
WRITER 2 IS WRITING
READER 2 is READING
READER 3 is READING
READER 0 is READING
READER 1 is READING
READER 4 is READING
WRITER 0 IS WRITING
WRITER 1 IS WRITING
WRITER 2 IS WRITING
READER 2 is READING
READER 4 is READING
READER 3 is READING
READER 1 is READING
READER 0 is READING
WRITER 0 IS WRITING
WRITER 1 IS WRITING
WRITER 2 IS WRITING
READER 4 is READING
READER 3 is READING
READER 0 is READING
READER 1 is READING
READER 2 is READING
WRITER 0 IS WRITING
```

WRITER 1 IS WRITING
WRITER 2 IS WRITING
READER 1 is READING
READER 2 is READING
READER 0 is READING
READER 4 is READING
READER 3 is READING

Experiment 5

Implementation of second readers writers problem

Description

In the first readers writers problem, readers can dominate the resource and it will not be possible for a writer to access the resource. In order to give preference to writers, it is necessary to prevent a subsequent reader process from gaining access to the shared resource till the writer accesses the shared resource and releases it. An algorithm to achieve this task is given below (Gary Nutt).

Here a stream of readers can enter the critical section till a writer arrives. When a writer arrives, it takes access of the resource after all existing readers leave the critical section. When the first writer arrives, it will obtain the *readBlock* semaphore. Then it blocks on the *writeBlock* semaphore, waiting for all readers to clear the critical section. The next reader to arrive will obtain the *writePending* semaphore and then block on the *readBlock* semaphore. If another writer arrives during this time, it will block on the *writeBlock* semaphore. If a second reader arrives, it will block on the *writePending* semaphore.

Algorithm

reader() {

```
while(TRUE) {
    <other computing>
    P(writePending);
    P(readBlock);
       P(mutex1);
      readCount = readCount +1;
      if(readCount == 1)
         P(writeBlock);
      V(mutex1);
    V(readBlock);
   V(writePending);


/* critical section */
access resource
```

```
   P(mutex1);
      readCount = readCount – 1;
      if(readCount == 0)
          V(writeBlock);
   V(mutex1);
  }
}
writer() {
  while(TRUE) {
    <other computing>;
   P(mutex2);
     writeCount = writeCount + 1;
     if(writeCount == 1)
     P(readBlock);
  V(mutex2);
     P(writeBlock);
/* critical section */
 access resource
     V(writeBlock);
 P(mutex2);
     writeCount = writeCount - 1;
     if(writeCount == 0)
     V(readBlock);
V(mutex2);
  }
}
```

The above algorithm is implemented using threads. *Mutex1*, mutex2, *writeBlock, readBlock and writePending* are the semaphores used.

Program

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
#include<stdlib.h>
void * reader (void *);
void * writer (void *);

sem_t mutex1, mutex2;
sem_t writeBlock;
sem_t readBlock;
sem_t writePending;
int readCount =0;
int writeCount =0;
main(int argc, char * argv[])
{
  int i, j, k;

  int N_readers, N_writers;
  int readers_num[100], writers_num[100];
  pthread_t tid_readers[100], tid_writers[100];
  printf("Enter the number of readers:");
  scanf("%d", &N_readers);

  printf("Enter the number of writers:");
  scanf("%d", &N_writers);

  for(k =0; k < N_readers; k++)
```

```
  readers_num[k] = k;
  for(k =0; k < N_writers; k++)
 writers_num[k] = k;
 if(sem_init(&mutex1, 0, 1) < 0)
  {
     perror("Could not init semaphore mutex1");
     exit(1);
  }
 if(sem_init(&mutex2, 0, 1) < 0)
  {
     perror("Could not init semaphore mutex2");
     exit(1);
  }
if(sem_init(&writeBlock, 0, 1) < 0)
  {
     perror("Could not init semaphore writeBlock");
     exit(1);
  }

 if(sem_init(&readBlock, 0, 1) < 0)
  {
     perror("Could not init semaphore readBlock");
     exit(1);
  }
 if(sem_init(&writePending, 0, 1) < 0)
  {
     perror("Could not init semaphore writePending");
     exit(1);
```

```
  }
for( i =0; i < N_readers; i++)
  {
     if(pthread_create(&tid_readers[i], NULL, reader,  &readers_num[i]  ))
     {
        perror("could not create reader thread");
        exit(1);
     }
  }
 for( j=0; j < N_writers; j++)
  {
    if(pthread_create(&tid_writers[j], NULL, writer, &writers_num[j]))
    {
       perror("could not create writer thread");
       exit(1);
    }
  }

  for (i =0; i < N_readers; i++)
  {
     pthread_join(tid_readers[i], NULL);
  }

  for ( j=0; j < N_writers; j++)
  {
    pthread_join(tid_writers[j], NULL);
  }
    sem_destroy(&mutex1);
```

```
  sem_destroy(&mutex2);
  sem_destroy(&readBlock);
  sem_destroy(&writeBlock);
  sem_destroy(&writePending);
}
void * reader (void* param)
{
    int i = *((int *) param);
     while(1)
      {
          sleep(1);
        if(sem_wait(&writePending) < 0)
         {
            perror("cannot decrement the semaphore writePending");
            exit(1);
         }
        if(sem_wait(&readBlock) < 0)
         {
            perror("cannot decrement the semaphore readBlock");
            exit(1);
         }
        if(sem_wait(&mutex1) < 0)
         {
            perror("cannot decrement the semaphore mutex1");
            exit(1);
         }
         readCount = readCount + 1;
         if(readCount == 1)
```

```
 {
    if(sem_wait(&writeBlock) < 0)
     {
       perror("cannot decrement the semaphore writeBlock");
       exit(1);
     }
 }
  if( sem_post(&mutex1) <0)
  {
     perror("cannot increment semaphore mutex1");
     exit(1);
  }
 if( sem_post(&readBlock) <0)
 {
     perror("cannot increment semaphore readBlock");
     exit(1);
 }
if( sem_post(&writePending) <0)
{
     perror("cannot increment semaphore writePending");
     exit(1);
}
          // READ RESOURCES
   printf("READER %d is READING \n", i);
     sleep(1);

if(sem_wait(&mutex1) < 0)
 {
```

```
            perror("cannot decrement the semaphore mutex");
            exit(1);
        }
        readCount = readCount - 1;
        if(readCount == 0)
         {
            if( sem_post(&writeBlock) < 0)
             {
                  perror("cannot increment semaphore mutex");
                  exit(1);
             }
          }
             if( sem_post(&mutex1) < 0)
             {
                  perror("cannot increment semaphore mutex");
                  exit(1);
             }
      }
}
void * writer (void * param)
{
   int i = *((int *) param);
   while(1)
    {
        sleep(1);
            if(sem_wait(&mutex2) < 0)
             {
                perror("cannot decrement the semaphore mutex2");
```

```
    exit(1);
  }
  writeCount = writeCount +1;
  if(writeCount == 1)
  {
    if(sem_wait(&readBlock) < 0)
     {
       perror("cannot decrement the semaphore readBlock");
       exit(1);
     }
  }

if( sem_post(&mutex2) < 0)
{
      perror("cannot increment semaphore mutex2");
      exit(1);
}
if(sem_wait(&writeBlock) < 0)
 {
    perror("cannot decrement the semaphore writeBlock");
    exit(1);
 }

 // WRITE RESOURCES
printf("WRITER %d IS WRITING \n", i);
if( sem_post(&writeBlock) < 0)
{
      perror("cannot increment semaphore writeBlock");
```

```
            exit(1);
        }
        if(sem_wait(&mutex2) < 0)
        {
          perror("cannot decrement the semaphore mutex2");
          exit(1);
        }
       writeCount = writeCount - 1;
       if(writeCount == 0)
       {

        if( sem_post(&readBlock) < 0)
        {
            perror("cannot increment semaphore readBlock");
            exit(1);
        }
       }
        if( sem_post(&mutex2) < 0)
        {
            perror("cannot increment semaphore mutex2");
            exit(1);
        }

   }

}
```

Output

```
anil@anil-300E4Z-300E5Z-300E7Z:~/NetworkLab$  ./expt5
```

Enter the number of readers:3

Enter the number of writers:2

READER 2 is READING
READER 0 is READING
READER 1 is READING
WRITER 0 IS WRITING
WRITER 1 IS WRITING
READER 2 is READING
READER 0 is READING
READER 1 is READING
WRITER 0 IS WRITING
WRITER 1 IS WRITING
READER 2 is READING
READER 0 is READING
READER 1 is READING
WRITER 0 IS WRITING
WRITER 1 IS WRITING
READER 2 is READING
READER 0 is READING
READER 1 is READING
WRITER 0 IS WRITING
WRITER 1 IS WRITING
READER 2 is READING
READER 0 is READING
READER 1 is READING
WRITER 0 IS WRITING
WRITER 1 IS WRITING
READER 2 is READING
READER 1 is READING
READER 0 is READING

Experiment 6

Inter-process Communication using Pipes, Message queues, and Shared Memory

Aim: To communicate between two processes using pipes and message queues.

Description:

In this experiment a pipe is used to connect a client with the server. The client reads the name of a file from the standard input. It then writes the name of the file on to the pipe. The server reads the file from the read end of the pipe. After that, the server opens the file and reads the contents of the file line by line. Each line that it reads is sent to the client and written to the standard output. The data is transferred using the message structure. It has a header, which gives the length of the message and the type of the message which is a positive integer. The data is carried in an array within the message structure. The data flow through pipes is as shown in the figure.

PICTURE

Algorithm

1. Create pipe 1 (fd1[0], fd1[1])
2. Create pipe 2 (fd2[0], fd2[1])
3. Fork a child process
4. Parent closes read end of pipe 1 (fd1[0])
5. Parent closes write end of pipe 2 (fd2[1])
6. Child closes write end of pipe 1 (fd1[1])
7. Child closes read end of pipe 2 (fd2[0])

Parent process (client process)

1. Read the filename from the standard input into the data portion of the message
2. Construct the message structure
3. Write the message to pipe 1
4. Read the message from the client on pipe2 and write to the standard output

Child process (server process)

1. Read the message sent by the client from pipe 1
2. Open the file
3. If there is an error, send an error message to the client
4. Otherwise, read each line from the file and send it as a message to the client on pipe 2

Program

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<string.h>
#include  "pipe.h"

void client(int, int);
void server(int, int);

int main(int argc, char ** argv)
 {
     int fd1[2], fd2[2]; // file descriptors for pipes
     pid_t childpid;

     // creation of pipes
     if(pipe(fd1) < 0)
      {
          perror("pipe creation error");
          exit(1);
       }

     if(pipe(fd2) < 0)
      {
          perror("pipe creation error");
          exit(1);
       }

     if((childpid = fork()) < 0)
      {
          perror("fork error");
          exit(1);
      }
      elseif(childpid == 0) // child process (server process)
       {
          close(fd2[0]); // child closes the read end of pipe 2
          close(fd1[1]); // child closes the write end of pipe 1

          server(fd1[0], fd2[1]);
          exit(0);
       }
       else   // parent process (client process)
        {
            close( fd1[0]);
            close(fd2[1]);
```

```
        client(fd2[0], fd1[1]);
        if(waitpid(childpid, NULL,0) < 0)
        {
              perror("waitpid error");
              exit(1);
         }
        exit(0);
      }
   }

void client(int readfd, int writefd)
 {
          int length;
          ssize_t n;
          struct message mesg;

           printf("Give the name of the file\n");
           fgets(mesg.message_data, MAXMESSAGEDATA, stdin);

           length = strlen(mesg.message_data);

          if(mesg.message_data[length -1] == '\n')
            length--;

          mesg.message_length = length;
          mesg.message_type = 1;

           // write message to the pipe
           write( writefd, &mesg, MESGHDRSIZE + mesg.message_length);

          // read from pipe and write to the standard output

          while(1)
           {
              if(n = read(readfd, &mesg, MESGHDRSIZE)) == -1)
               {
                      perror("read error");
                       exit(1);
                }

              if( n!= MESGHDRSIZE)
               {
                    fprintf(stderr, "header size not same");
                    exit(1);
                }
```

```
                length = mesg.message_length;
                if(length == 0) break;

                n = read(readfd, mesg.message_data, length);
                write(STDOUT_FILENO, mesg.message_data, n);
              }
            }

void server(int readfd, int writefd)
 {
        FILE * fp;
        ssize_t n;
        struct message mesg;
        size_t length;

        mesg.message_type = 1;
        n = read(readfd, &mesg, MESGHDRSIZE);

        if( n!=MESGHDRSIZE)
         {
              fprintf(stderr, "header size not same \n");
              exit(1);
         }

       length =mesg.message_length;
       n = read(readfd, mesg.message_data, length);
       mesg.message_data[n] = '\0';

       if( (fp = fopen(mesg.message_data, "r")) ==NULL)
         {
           snprintf(mesg.message_data + n, sizeof(mesg.message_data) -n, ":cant open\n");
           mesg.message_length = strlen(mesg.message_data);
           write(writefd, &mesg, MESGHDRSIZE + mesg.message_length);
         }
          else
          {
              while(fgets(mesg.message_data, MAXMESSAGEDATA, fp) != NULL)
                {
                    mesg.message_length = strlen(mesg.message_data);
                    mesg.message_type = 1;
                    write(writefd, &mesg, MESGHDRSIZE + mesg.message_length);
                }

                fclose(fp);
          }
       mesg.message_length = 0;
```

```
        write(writefd, &mesg, MESGHDRSIZE + mesg.message_length);
    }
```

Header file

```
#ifndef _PIPE
#define _PIPE
#include<stdio.h>
#include<limits.h>

#define MAXMESSAGEDATA(PIPE_BUF -2*sizeof(long)) // PIPE_BUF is
the maximum amount of data that can be written to a pipe

#define MESGHDRSIZE (sizeof(struct message) – MAXMESSAGEDATA)

struct message {
                long message_length;
                long message_type;
                char message_data[MAXMESSAGEDATA];
        };


#endif
```

Output

Using pipes send a filename from a client to a server. Open the file in the server and send the contents of the file to the client using pipe and display it on the console.