2017

SOURCE CODE -PA1

DESIGN

VIKRAM G AND MOUNA GIRI

1. CPU BENCHMARKING

FILE NAME – threaddemo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>
void *IOPS_function(void *ptr);
void *FLOPS_function(void *ptr);


struct timeval starttime, endtime;
int main()
{
pthread_t threads[50];

int iret1,iret2,i,j;
printf("CPU performance benchmark:\n");
for(i=1;i<10;i=i*2)
{
        long int filesize=(100000)/i;

gettimeofday(&starttime,NULL);
double start_time = 1000000 * starttime.tv_sec + starttime.tv_usec;
double sec1 = start_time / 1000000.0;

long int *param = (long int *)malloc(100000* sizeof(int));
        param[0] = filesize;
for(j=0;j<i;j++)
{
iret1 = pthread_create( &threads[j], NULL, IOPS_function, param);
sleep(0);
}

for(j=0;j<i;j++)
{
iret1 = pthread_join(threads[j],NULL);
```

```
}
gettimeofday(&endtime,NULL);
double end_time = 1000000 * endtime.tv_sec + endtime.tv_usec;
double sec2 = end_time / 1000000.0;
double total_duration=(double)sec2-sec1;
double result=(double)((100000/total_duration)*14)/1000000000;
printf("The processor speed in terms of GigaIOPS with %d  threads is %1f\n",i,result );
free(param);
}

printf("\n\n");
for(i=1;i<10;i=i*2)
{

        long int fsize=(100000)/i;

gettimeofday(&starttime,NULL);
double start_time = 1000000 * starttime.tv_sec + starttime.tv_usec;
double sec1 = start_time / 1000000.0;

long int *param1 = (long int *)malloc(100000* sizeof(int));
        param1[0] = fsize;

for(j=0;j<i;j++)
{
iret1 = pthread_create( &threads[j], NULL, FLOPS_function, param1);
sleep(0);
}

for(j=0;j<i;j++)
{
iret1 = pthread_join(threads[j],NULL);
}
gettimeofday(&endtime,NULL);
double end_time = 1000000 * endtime.tv_sec + endtime.tv_usec;
double sec2 = end_time / 1000000.0;
double total_duration=(double)sec2-sec1;
double result=(double)((100000/total_duration)*14)/1000000000;
printf("The processor speed in terms of GigaFLOPS with %d  threads is %1f\n",i,result );
free(param1);
}
```

```
}



void *IOPS_function(void *parm )
{
long int *param = (long int *)parm;
   long int fsize = param[0];
long int i,sum;
for       (i=0;i<fsize;i++)
{
sum=100+200+500*500+80*90+71+69+411+987+654+321*458*125;
}
}

void *FLOPS_function(void *parm )
{
long int *param = (long int *)parm;
   long int fsize = param[0];
long int i;
double sum;
for       (i=0;i<fsize;i++)
{
sum=10.225+20.669+50.587*50.17+80.698*90.89+71.12+69.13+41.2+98.025+65.6+21.
45*458.12*12.789;
}

}
```

2. MEMORY BENCHMARKING

FILE NAME – memory.c

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

```
#include<sys/time.h>
#include<string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <malloc.h>


char *data1,*data2;
struct timeval starttime, endtime;




void *read_write_function(void *parm)
{
int i;
        long int *param = (long int *)parm;
   long int fsize = param[0];
        int size=param[1];
        int count=param[2];

        //printf("%d\t,%d\t,%d\n",fsize,size,count);

for( i=0;i<fsize/size;i++)
{
memcpy(&data1[fsize*count]+(size*i),&data2[fsize*count]+(size*i),size);
}



}

void *sequential_write_function(void *parm)
{

int i;
        long int *param = (long int *)parm;
   long int fsize = param[0];
        int size=param[1];
        int count=param[2];

for(i=0;i<fsize/size;i++)
```

```
{

memset(&data2[fsize*count]+(size*i),'g',size);
}
}

void *random_write_function(void *parm)
{

int i;
        long int *param = (long int *)parm;
   long int fsize = param[0];
        int size=param[1];
        int count=param[2];
        long int newsize=fsize/size;

for(i=0;i<newsize;i++)
{
long int
long rand=random()%(newsize-1);
memset(&data2[fsize*count]+(size*rand),'g',size);
}
}




void main()
{
pthread_t threads[15];
int bsize,size;
int k,p;
printf("Memory benchmarking\n");
printf("Select the block size : Enter 1 for 8B,2 for 8KB, 3  for 8MB, 4 for 80MB\n");

scanf("%d",&bsize);
printf("\n");
if(bsize==1)
{
size=(int)8;
}
```

```
else if (bsize==2)
{
        size=(int)8*1024;
}
else if(bsize==3)
{
        size=(int)8*1048576;
}

else if(bsize==4)
{
        size=(long int) 8*10485760;
}
else {
printf("invalid option try again\n");
exit(0);
}

printf("sequential read+write memory access using different number of threads and
their latency and throughput\n");

printf("memory function for size %d\n",size);
for(k=1;k<10;k=k*2)
{
        data1=malloc(1024*1024*1100);
        data2=malloc(1024*1024*1100);
        memset(data2,'g',1024*1024*1100);

        long int filesize=(1024*1024*1100)/k;

        gettimeofday(&starttime,NULL);
        double start_time = 1000000 * starttime.tv_sec + starttime.tv_usec;
        double sec1 = start_time / 1000000.0;
        int count=0;
        long int *param = (long int *)malloc(1024*1024*1100);
        param[0] = filesize;
        param[1] = size;

        for(p=0;p<k;p++)
        {
                param[2]=count;
```

```
                count=count+1;
                pthread_create(         &threads[p],NULL,read_write_function,param);
sleep(0);
        }

        for(p=0;p<k;p++)
        {
   pthread_join(threads[p],NULL);

        }
        count=0;
gettimeofday(&endtime,NULL);
double end_time = 1000000 * endtime.tv_sec + endtime.tv_usec;
double sec2 = end_time / 1000000.0;
double total_duration=(double)sec2-sec1;

float throughput=(float)((1024*1024*1100)/(total_duration*1024*1024));


printf("Throughput of memory with %d threads is %1f\n",k,throughput);


if(size==8)
{
float latency= (float)(total_duration*1000)/(1024*1024*1100);
printf("latency of memory with %d threads is %1f\n\n",k,latency);
}


free(data1);
free(data2);
free(param);

}
printf("\n\n");

printf("sequential write memory access using different number of threads and their
latency and throughput\n");
printf("\n\n");
```

```
printf("memory function for size %d\n",size);
for(k=1;k<10;k=k*2)
{

        data2=malloc(1024*1024*1100);
long int filesize=(1024*1024*1100)/k;

        gettimeofday(&starttime,NULL);
double start_time = 1000000 * starttime.tv_sec + starttime.tv_usec;
double sec1 = start_time / 1000000.0;
int count=0;
        long int *param = (long int *)malloc(1024*1024*1100);
        param[0] = filesize;
        param[1] = size;

        for(p=0;p<k;p++)
        {
                param[2]=count;
                count = count+1;
        pthread_create(         &threads[p],NULL,sequential_write_function,param);
        sleep(0);
        }

        for(p=0;p<k;p++)
        {
   pthread_join(threads[p],NULL);

        }
gettimeofday(&endtime,NULL);
double end1_time = 1000000 * endtime.tv_sec + endtime.tv_usec;
double sec2 = end1_time / 1000000.0;
double total_duration=(double)sec2-sec1;

float throughput=(float)((1024*1024*1100)/(total_duration*1024*1024));


printf("Throughput of memory with %d threads is %1f\n",k,throughput);


if(size==8)
{
```

```
float latency= (float)(total_duration*1000)/(1024*1024*1100);
printf("latency of memory with %d threads is %1f\n\n",k,latency);
}


free(data2);

}
printf("\n\n");

printf("random write memory access using different number of threads and their
latency and throughput\n");
printf("\n\n");

printf("memory function for size %d\n",size);
for(k=1;k<10;k=k*2)
{

        data2=malloc(1024*1024*1100);
long int filesize=(1024*1024*1100)/k;

        gettimeofday(&starttime,NULL);
double start_time = 1000000 * starttime.tv_sec + starttime.tv_usec;
double sec1 = start_time / 1000000.0;
int count=0;
        long int *param = (long int *)malloc(1024*1024*1100 );
        param[0] = filesize;
        param[1] = size;

        for(p=0;p<k;p++)
        {
                param[2]=count;
                count = count+1;
        pthread_create(         &threads[p],NULL,random_write_function,param);
        sleep(0);
        }

        for(p=0;p<k;p++)
        {
   pthread_join(threads[p],NULL);
```

```
        }
gettimeofday(&endtime,NULL);
double end1_time = 1000000 * endtime.tv_sec + endtime.tv_usec;
double sec2 = end1_time / 1000000.0;
double total_duration=(double)sec2-sec1;

float throughput=(float)((1024*1024*1100)/(total_duration*1024*1024));


printf("Throughput of memory with %d threads is %1f\n",k,throughput);


if(size==8)
{
float latency= (float)(total_duration*1000)/(1024*1024*1100);
printf("latency of memory with %d threads is %1f\n\n",k,latency);
}


free(data2);

}
printf("\n\n");

}
```

3. DISK BENCHMARKING

FILE NAME – DiskTask_Random.java

```
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Random;
import java.util.Scanner;
```

```
public class DiskTask_Random extends Thread{
        Random r = new Random();
         static Thread thread_numb;
         static int commandLine;
         private static String inputFile = "largeFile.txt";
         private static String copiedFile = "Result.db";
         static File fileName = new File(inputFile);
         static double new_file_size;
         static int count;
         static double time_taken ;
         static double sum;
         static double sUM_TT;

        public static void main(String args[]) throws InterruptedException {

                DiskTask_Random[] run = new DiskTask_Random[8];
                 Scanner console = new Scanner (System.in);
            while (true) {
              System.out.print("Please enter the blocksize[8b, 80b, 8M , 80Mb and 0-
quit] ");

              commandLine = console.nextInt();
              if (commandLine == 0)  {
                 break;
              }

                        System.out.println("\n" + commandLine + " is the Block Size");


    int[] threadCount={1,2,4,8};
    for(int a=0;a<threadCount.length;a++){
        {
                //Thread.sleep(1000);
                int no_of_thread=threadCount[a];
        System.out.println("No of threads running is "+no_of_thread);

        double file_Size = fileName.length();

         new_file_size=file_Size/no_of_thread;

         count=0;
```

```
        sum=0;
        time_taken=0;
        sUM_TT=0;

    for(int i=0;i<(no_of_thread);i++){

        run[i] = new DiskTask_Random();
        run[i].start();
         count=i;

    }
    for(int c=0;c<(no_of_thread);c++)
                run[c].join();

         sum=sUM_TT;
                if (commandLine == 8)
                {

                        sum=sum/1000000.0;
                        double result_latency=(sum/file_Size);
        System.out.println("Latency for random Read"+ result_latency + " ms");
                }
                else
                {
                        //System.out.println("sum for final"+sum);

                        sum=sum/1000000000.0;

                        double result_throughput=(file_Size/1048576)/sum;
                        System.out.println("Throughput for Random Read"+
result_throughput + " MB/sec");
                }


        }
        }
            }
}
```

```
        public  void random_read(double file_size, int counter, int block_size) throws
IOException
                {

                 RandomAccessFile rafIN = new RandomAccessFile(inputFile, "r");
                        byte[] byteArray = new byte[block_size];
                        rafIN.seek((long) (file_size*counter));
                        int value=(int)file_size/block_size;

                        double start = System.nanoTime();


                        for (int i = 0; i < value; i++) {

                           int ran=r.nextInt((int)file_size);

                           rafIN.seek(ran);
                           rafIN.read(byteArray);

                        }
                        time_taken = System.nanoTime()- start;


                        sUM_TT +=time_taken;
                        rafIN.close();

        }



        @Override
        public void run()  {

                        try {

                                random_read(new_file_size,count,commandLine);
                        } catch (IOException e) {
```

```
                    e.printStackTrace();
                }
            }

}
```

b. FILE NAME : DiskTask_Sequential.java

```
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Scanner;


public class DiskTask_Sequential extends Thread{

    static Thread thread_numb;
    static int commandLine;
    private static String inputFile = "largeFile.txt";
    private static String copiedFile = "Result.db";
    static File fileName = new File(inputFile);
    static double new_file_size;
    static int count;
    static double time_taken ;
    static double sum;
    static double sUM_TT;

   public static void main(String args[]) throws InterruptedException {

        DiskTask_Sequential[] run = new DiskTask_Sequential[8];
         Scanner console = new Scanner (System.in);
      while (true) {
```

```
            System.out.print("Please enter the blocksize[8b, 80b, 8M , 80Mb and 0-
quit] ");

            commandLine = console.nextInt();
            if (commandLine == 0)  {
               break;
            }

                        System.out.println("\n" + commandLine + " is the Block Size");


      int[] threadCount={1,2,4,8};
      for(int a=0;a<threadCount.length;a++){
         {
               //Thread.sleep(1000);
               int no_of_thread=threadCount[a];
                Thread thread_numb;
         System.out.println("No of threads running is "+no_of_thread);

         double file_Size = fileName.length();

          new_file_size=file_Size/no_of_thread;

          count=0;
         sum=0;
         time_taken=0;
         sUM_TT=0;

      for(int i=0;i<(no_of_thread);i++){

         run[i] = new DiskTask_Sequential();
         run[i].start();
          count=i;

      }
      for(int c=0;c<(no_of_thread);c++)
                run[c].join();

          sum=sUM_TT;
               if (commandLine == 8)
               {
```

```
                        sum=sum/1000000.0;
                        double result_latency=(sum/file_Size);
        System.out.println("Latency for Seq Read "+ result_latency + " ms");
                }
                else
                {

                        sum=sum/1000000000.0;

                        double result_throughput=(file_Size/1048576)/sum;
                        System.out.println("Throughput for Seq Read "+
result_throughput + " MB/sec");
                }


        }
        }
            }
}


         public  void seq_read(double file_size, int counter, int block_size) throws
IOException
        {

                RandomAccessFile rafIN = new RandomAccessFile(inputFile, "r");
                byte[] byteArray = new byte[block_size];
                rafIN.seek((long) (file_size*counter));
                int value=(int)file_size/block_size;
                double start = System.nanoTime();

                for (int i = 0; i < value; i++) {

                        rafIN.seek(rafIN.getFilePointer());
                   rafIN.read(byteArray);
                }

                time_taken = System.nanoTime()- start;
```

```
                sUM_TT +=time_taken;
                rafIN.close();
        }



        @Override
        public void run()  {

                        try {

                                seq_read(new_file_size,count,commandLine);
                        } catch (IOException e) {

                                e.printStackTrace();
                        }
                }


}
```

c. FILE NAME : DiskTask_ReadnWrite.jav

```
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;

import java.util.Scanner;



public class DiskTask_ReadnWrite extends Thread{

            static Thread thread_numb;
            static int commandLine;
            private static String inputFile = "largeFile.txt";
            private static String copiedFile = "Result.db";
            static File fileName = new File(inputFile);
```

```
        static double new_file_size;
        static int count;
        static double time_taken ;
        static double sum;
        static double sUM_TT;

    public static void main(String args[]) throws InterruptedException {

            DiskTask_ReadnWrite[] run = new DiskTask_ReadnWrite[8];
             Scanner console = new Scanner (System.in);
          while (true) {
            System.out.print("Please enter the blocksize[8b, 80b, 8M , 80Mb and 0-
quit] ");

            commandLine = console.nextInt();
            if (commandLine == 0)  {
               break;
            }

                    System.out.println("\n" + commandLine + " is the Block Size");


  int[] threadCount={1,2,4,8};
  for(int a=0;a<threadCount.length;a++){
      {

            int no_of_thread=threadCount[a];
      System.out.println("No of threads running is "+no_of_thread);

      double file_Size = fileName.length();

       new_file_size=file_Size/no_of_thread;

      count=0;
      sum=0;
      time_taken=0;
      sUM_TT=0;

  for(int i=0;i<(no_of_thread);i++){

      run[i] = new DiskTask_ReadnWrite();
```

```
            run[i].start();
             count=i;

        }
        for(int c=0;c<(no_of_thread);c++)
                        run[c].join();

             sum=sUM_TT;
                    if (commandLine == 8)
                    {
                            System.out.println("sum for final"+sum);
                            sum=sum/1000000.0;
                            double result_latency=(sum/file_Size);
            System.out.println("Latency for Read and Write : "+ result_latency + " ms");
                    }
                    else
                    {

                            sum=sum/1000000000.0;

                            double result_throughput=(file_Size/1048576)/sum;
                            System.out.println("Throughput for Read and Write  :"+
result_throughput + " MB/sec");
                    }


            }
            }
                }
}







             public  void read_write(double file_size, int counter, int block_size) throws
IOException
                    {
```

```
                        RandomAccessFile rafIN = new RandomAccessFile(inputFile, "rw");
                        RandomAccessFile rafout = new RandomAccessFile(copiedFile,
"rw");
                        byte[] byteArray = new byte[block_size];
                        rafIN.seek((long) (file_size*counter));
                        int value=(int)file_size/block_size;

                        double start = System.nanoTime();
                        for (int i = 0; i < value; i++) {
                                int currentPosition = (int)rafIN.getFilePointer();
                           rafIN.read(byteArray);
                           rafout.seek(currentPosition);
                           rafout.write(byteArray);

                        }
                        time_taken = System.nanoTime()- start;


                        sUM_TT +=time_taken;
                        rafIN.close();
                        rafout.close();
                }




        @Override
        public void run()  {

                        try {
                                //System.out.println("Disk Metrics Calculations for"+
commandLine + "bytes");
                                read_write(new_file_size,count,commandLine);
                        } catch (IOException e) {

                                e.printStackTrace();
                        }
                }
```

```
}
```

4. NETWORK BENCHMARKING

FILENAME:net_server.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#define PORT 8080


int main(int argc, char const *argv[])
{
   int server_fd, new_socket, valread;
   struct sockaddr_in address;
   int opt = 1;
   int addrlen = sizeof(address);
   int LENGTH=1024;
          char buffer[LENGTH];


   FILE *fp;


   if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
   {
      perror("socket failed");
      exit(EXIT_FAILURE);
   }


   address.sin_family = AF_INET;
   address.sin_addr.s_addr = INADDR_ANY;
   address.sin_port = htons( PORT );

   printf("waiting for connection\n");
```

```
if (bind(server_fd, (struct sockaddr *)&address,sizeof(address))<0)
{
   perror("bind failed");
   exit(EXIT_FAILURE);
}
if (listen(server_fd, 3) < 0)
{
   perror("listen");
   exit(EXIT_FAILURE);
}
if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
               (socklen_t*)&addrlen))<0)
{
   perror("accept");
   exit(EXIT_FAILURE);
}

        printf("connection successfull\n");



        fp=fopen("downloadedfile.txt","a");

         bzero(buffer, LENGTH);
      int fr_block_sz = 0;


      while((fr_block_sz = recv(new_socket, buffer, LENGTH, 0)) > 0)
      {
         int write_sz = fwrite(buffer, sizeof(char), fr_block_sz, fp);
         if(write_sz < fr_block_sz)
         {
            error("File write failed on server.\n");
         }
         bzero(buffer, LENGTH);
         if (fr_block_sz == 0 || fr_block_sz != 1024)
         {
            break;
         }
      }
```

```
   fclose(fp);

   return 0;
}
```

FILENAME: net_client.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <pthread.h>
#define PORT 8080

struct timeval starttime, endtime;

  void *threadfunc(void *parm)
{
           long int i;
long int LENGTH=1024;


          char buffer[LENGTH];
long int *param = (long int *)parm;
   long int fsize = param[0];
          int sock=param[1];
                FILE *fp=param[2];



bzero(buffer, LENGTH);
      int fs_block_sz;

                                    long int k=(fsize/LENGTH);
```

```
                    for(i=0;i<k;i++)
    {   fs_block_sz = fread(buffer, sizeof(char), LENGTH, fp);
        if(send(sock, buffer, fs_block_sz, 0) < 0)
        {
            fprintf(stderr, "ERROR: Failed to send file \n");
            break;
        }
        bzero(buffer, LENGTH);
    }




}
int main(int argc, char const *argv[])
{
         FILE *fp;
  struct sockaddr_in address;
  int sock = 0,i,j;
         int num;
  struct sockaddr_in serv_addr;


         pthread_t p[50];

  if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    printf("\n Socket creation error \n");
    return -1;
  }

  memset(&serv_addr, '0', sizeof(serv_addr));

  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(PORT);


  if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)
  {
    printf("\nInvalid address/ Address not supported \n");
    return -1;
```

```
   }

   if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
   {
      printf("\nConnection Failed \n");
      return -1;
   }



         fp = fopen ("new.txt", "r");
         fseek(fp, 0L, SEEK_END);
long int size = ftell(fp);

rewind(fp);
 printf("connected to server\n");
         printf("please enter the number of threads you wish to run in between[1,2,4,8]
:\n");
         scanf("%d",&num);
         printf("\n");


long int fsize=size/num;

long int *param = (long int *)malloc(1024*1024);
         param[0] = fsize;

         param[1]=sock;
         param[2]=fp;

gettimeofday(&starttime,NULL);
         double start_time = 1000000 * starttime.tv_sec + starttime.tv_usec;
         double sec1 = start_time / 1000000.0;

for(j=0;j<num;j++)
{

         pthread_create(&p[j],NULL,threadfunc,param );

}
```

```
for(j=0;j<num;j++)
{

        pthread_join(p[j],NULL);

}

gettimeofday(&endtime,NULL);
double end_time = 1000000 * endtime.tv_sec + endtime.tv_usec;
double sec2 = end_time / 1000000.0;
double total_duration=(double)sec2-sec1;

float throughput=(float)((size)/(total_duration*1024*1024));

printf("Throughput of client for uploading file with %d threads is
%1f\n",num,throughput);

float latency= (float)(total_duration*1000)/(size);
printf("latency of client for uploading  with %d threads is %1f\n\n",num,latency);



fclose(fp);

free(param);


   return 0;
}
```