CS 553 Cloud Computing
Programming Assignment 1
Hariprasad Ravi Kumar
(A20348609)

SOURCE CODE

CPU Benchmarking: (C)

CPUBenchmark.c

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<sys/time.h>
#include<pthread.h>
#include<string.h>

#define numsec 1
#define ITERATIONS 10000000

FILE *fptr;

void *threadFunctionFlop(void *arg);
void *threadFunctionIops(void *arg);


void flops(int numberOfThreads)
{
	clock_t start, start1, end, end1;
	double cpu_time_used;
	int n,i,count=0;
	long double a=5;


	time_t lasttime, thistime;
	struct timezone tzp;
	pthread_t th[10];// array of threads

	long iterations=ITERATIONS/numberOfThreads;
	char iterationStr[20];
	snprintf(iterationStr, 20, "%lu",iterations);

	printf("\nProgram to find FLOPS for %d threads",numberOfThreads);
	start = clock();
	for(n=0;n<numberOfThreads;n++)
	{
		pthread_create(&th[n],NULL,threadFunctionFlop,iterationStr);
		pthread_join(th[n], NULL);
	}
	end = clock();
	cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

```
        //printf("\nTime: %f ms\n",cpu_time_used);
        double Flops=(ITERATIONS)/(double)(cpu_time_used);
        double gFlops=(double)Flops/1000000000;// Calculate Giga Flops Formula: Flops *
10raised to (-9).

        printf("\nGFLOPS : %f\n",gFlops);


                if(numberOfThreads==4)
                {

                  fptr=(fopen("600samplesFLOPS.txt","w+"));
                  if(fptr==NULL){
                     printf("Error!");
                     exit(1);
                  }
                lasttime = time(NULL);
                for(i=1;i<601;i++){
                while(1){
               thistime = time(NULL);
               if(thistime - lasttime >= numsec)
                  break;
               if(thistime - lasttime >= 2)
                  sleep(thistime - lasttime - 1);
                      }

                for(n=0;n<numberOfThreads;n++)
                {
                        pthread_create(&th[n],NULL,threadFunctionFlop,iterationStr);
                        pthread_join(th[n], NULL);
                }
        count = count +1;
        //printf("%d",count);
        end1 = clock();
        cpu_time_used = ((double) (end1 - start1)) / CLOCKS_PER_SEC;

        //printf("\nTime: %f ms\n",cpu_time_used);
        double Flops=(ITERATIONS)/(double)(cpu_time_used);
        double gFlops=(double)Flops/1000000000;// Calculate Giga Flops Formula: Flops *
10raised to (-9).

        //printf("\nGFLOPS : %f\n",gFlops);

     lasttime += numsec;     /* update lasttime */
        fprintf(fptr,"%e \n",gFlops);
        }
```

```
        fclose(fptr);
 }
}

void iops(int numberOfThreads)
{
        clock_t start, start1, end, end1;
        int i,n,count=0;
        int a=5;
        double cpu_time_used;
        time_t lasttime, thistime;

        //struct timeval start, end;
        struct timezone tzp;
        pthread_t th[10];// array of threads

        long iterations=ITERATIONS/numberOfThreads;
        char iterationStr[20];
        snprintf(iterationStr, 20, "%lu",iterations);

        printf("\nProgram to find IOPS for %d threads",numberOfThreads);
        start=clock();
        for(i=0;i<numberOfThreads;i++)
        {
                pthread_create(&th[i],NULL,threadFunctionIops,iterationStr);
                pthread_join(th[i], NULL);
        }
        end = clock();
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
        //printf("\nTime: %f ms\n",cpu_time_used);
        double Iops=(ITERATIONS)/(double)(cpu_time_used);
        double gIops=(double)Iops/1000000000;

        printf("\nGIOPS : %f\n",gIops);

        if(numberOfThreads==4)
        {

          fptr=(fopen("600samplesIOPS.txt","w+"));
          if(fptr==NULL){
             printf("Error!");
             exit(1);
          }


                lasttime = time(NULL);
```

```
                for(i=1;i<601;i++){
                while(1){
               thistime = time(NULL);
               if(thistime - lasttime >= numsec)
                 break;
               if(thistime - lasttime >= 2)
                 sleep(thistime - lasttime - 1);
                 }

                start1 = clock();
                for(n=0;n<numberOfThreads;n++)
                {
                        pthread_create(&th[n],NULL,threadFunctionIops,iterationStr);
                        pthread_join(th[n], NULL);
                }
        count = count +1;
        //printf("%d",count);
        end1 = clock();
        cpu_time_used = ((double) (end1 - start1)) / CLOCKS_PER_SEC;

        //printf("\nTime: %f ms\n",cpu_time_used);
        double Iops=(ITERATIONS)/(double)(cpu_time_used);
        double gIops=(double)Iops/1000000000;

        //printf("GIOPS : %f\n",gIops);

    lasttime += numsec;     /* update lasttime */
        fprintf(fptr,"%e \n",gIops);
        }
        fclose(fptr);
 }
}




int main()
{
        int numberOfThreads;;
        while(1)
        {
                printf("\nEnter the no. of threads:(1/2/4) (Exit-0) : ");
                scanf("%d",&numberOfThreads);
                if(numberOfThreads!=1 && numberOfThreads!=2 && numberOfThreads!=4
&& numberOfThreads!=0 )
                {
```

```
                printf("\nInvalid thread. Please enter again");

            }
            else if(numberOfThreads==0)
            {
                exit(0);
            }
            else
            {
                flops(numberOfThreads);
                iops(numberOfThreads);
            }
        }

        return 0;
}


// To calculate Flops using thread function
void *threadFunctionFlop(void *arg)
{
        int n;
        double sum=5.5;

        long iterations=strtol((char*)arg,NULL,0); //  converting string argument to long
        for(n = 0; n < iterations; n++)
        {
        sum=sum+sum;
        }
        return NULL;
}

// To calculate Iops using thread function
void *threadFunctionIops(void *arg)
{
        int n;
        int sum=5;

        long iterations=strtol((char*)arg,NULL,0); //  converting string argument to long
        for(n = 0; n < iterations; n++)
        {
        sum=sum+sum;
        }
        return NULL;
}
```

Theory_performance.java

```
import java.io.*;

public class Theory_performance
{
private static void core_speed() throws Exception
{
String[] cmd1 = {"/bin/sh","-c","cat /proc/cpuinfo | grep processor | wc -l"};;
String[] cmd2 = {"/bin/sh","-c","cat /proc/cpuinfo | grep 'GHz'"};;
Runtime rt = Runtime.getRuntime();
Process proc1,proc2;
proc1 = rt.exec(cmd1);
proc2 = rt.exec(cmd2);
BufferedReader stdInput = new BufferedReader(new
InputStreamReader(proc1.getInputStream()));
String Cores = null;
Cores = stdInput.readLine();
int num_cores = Integer.parseInt(Cores);
stdInput = new BufferedReader(new InputStreamReader(proc2.getInputStream()));
String speed = null;
speed = stdInput.readLine();
speed = (String) speed.subSequence(speed.length()-7, speed.length()-3);
float speed_ghz = Float.parseFloat(speed);
System.out.println("Theoritical Performance Of Your CPU In GFLOPS:
"+speed_ghz*num_cores*4+"\n");
}

public static void main(String[] args) throws Exception
{
core_speed();
}
}
```

DISK Benchmarking: (C)

DiskBenchmark.c

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<sys/time.h>
#include<string.h>
#include<pthread.h>

#define BYTE 1
#define KILOBYTE 1024
#define MEGABYTE 1024*1024

#define BITERATIONS 10000000
#define KBITERATIONS 1000
#define MBITERATIONS 10


// 1B block size

void *fileWriteByteSequential()
{
	FILE *fp;
	char c='c';
	int i;
	fp = fopen("file.txt", "w+");
	for(i=0;i<BITERATIONS;i++)
	{
		fputc(c,fp);
	}
	fclose(fp);
}

void *fileWriteByteRandom()
{
	FILE *fp;
	char c='c';
	int i;
	fp = fopen("file.txt", "w+");
	//char bufferB[BYTE];
	for(i=0;i<BITERATIONS;i++)
	{
		fputc(c,fp);
	}
	fclose(fp);
}

void *fileReadByteSequential()
```

```
{
	FILE *fp;
	int i;
	char bufferB[BYTE];
	fp = fopen("file.txt", "r+");
	for(i=0;i<BITERATIONS;i++)
	{
		fread(bufferB, 1, BYTE, fp);
	}
	fclose(fp);
}

void *fileReadByteRandom()
{
	FILE *fp;
	char c[]="c";
	int i;
	char bufferB[BYTE];
	fp = fopen("file.txt", "r+");
	for(i=0;i<BITERATIONS;i++)
	{
		fread(bufferB, 1, BYTE, fp);
	}
	fclose(fp);
}


// 1KB block size

void *fileWriteKiloByteSequential()
{
	FILE *fp;
	int i,j;
	char c[KILOBYTE];
	for(j=0;j<KILOBYTE;j++)
	{
		c[j]='m';
	}

	fp = fopen("file.txt", "w+");
	//char bufferKb[KILOBYTE];
	for(i=0;i<KBITERATIONS;i++)
	{
		fwrite(c, 1, KILOBYTE, fp);
	}

	fclose(fp);
}

void *fileWriteKiloByteRandom()
```

```
{
	FILE *fp;
	int i,j;
	char c[KILOBYTE];
	for(j=0;j<KILOBYTE;j++)
	{
		c[j]='m';
	}
	fp = fopen("file.txt", "w+");
	char bufferKb[KILOBYTE];
	for(i=0;i<KBITERATIONS;i++)
	{
		int r=rand()%KILOBYTE;
		fseek(fp,r,SEEK_SET);
		fwrite(c, 1, KILOBYTE, fp);
	}

	fclose(fp);
}

void *fileReadKiloByteSequential()
{
	FILE *fp;
	int i;
	fp = fopen("file.txt", "r+");
	char bufferKb[KILOBYTE];
	for(i=0;i<KBITERATIONS;i++)
	{
		fread(bufferKb, KILOBYTE, 1, fp);
	}
	fclose(fp);
}

void *fileReadKiloByteRandom()
{
	FILE *fp;
	int i;
	fp = fopen("file.txt", "r+");
	char bufferKb[KILOBYTE];
	for(i=0;i<KBITERATIONS;i++)
	{
		int r=rand()%KILOBYTE;
		fseek(fp,r,SEEK_SET);
		fread(bufferKb, KILOBYTE, 1, fp);
	}
	fclose(fp);
}
```

```
// 1MB block size

void *fileWriteMegaByteSequential()
{
	FILE *fp;
	int i,j;
	char c[MEGABYTE];
	for(j=0;j<MEGABYTE;j++)
	{
		c[j]='m';
	}
	fp = fopen("file.txt", "w+");
	char bufferKb[MEGABYTE];
	for(i=0;i<MBITERATIONS;i++)
	{
		fwrite(c, 1, MEGABYTE, fp);
	}
	fclose(fp);
}

void *fileWriteMegaByteRandom()
{
	FILE *fp;
	int i,j;
	char c[MEGABYTE];
	for(j=0;j<MEGABYTE;j++)
	{
		c[j]='m';
	}
	fp = fopen("file.txt", "w+");
	char bufferKb[MEGABYTE];
	for(i=0;i<MBITERATIONS;i++)
	{
		int r=rand()%MEGABYTE;
		fseek(fp,r,SEEK_SET);
		fwrite(c, 1, MEGABYTE, fp);
	}

	fclose(fp);
}

void *fileReadMegaByteSequential()
{
	FILE *fp;
	int i;
	fp = fopen("file.txt", "r+");
	char bufferKb[MEGABYTE];
	for(i=0;i<MBITERATIONS;i++)
	{
		fread(bufferKb, MEGABYTE, 1, fp);
```

```
	}
	fclose(fp);
}

void *fileReadMegaByteRandom()
{
	FILE *fp;
	int i;
	fp = fopen("file.txt", "r+");
	char bufferKb[MEGABYTE];
	for(i=0;i<MBITERATIONS;i++)
	{
		int r=rand()%MEGABYTE;
		fseek(fp,r,SEEK_SET);
		fread(bufferKb, MEGABYTE, 1, fp);
	}
	fclose(fp);
}


void main()
{
	struct timeval start, end;
	struct timezone tzp;
	clock_t startTime, endTime;
	double timeDiff,latency,throughput;

	printf("\nProgram to find Disk Benchmark\n.....\n.....");
	pthread_t th[10];// array of threads
	int i;
	int ch,nthread;
	while(1)
	{

	printf("\n\nEnter the Block Size:\n1.BYTE\n2.KILOBYTE\n3.MEGABYTE\n4.EXIT : \n");
	scanf("%d",&ch);
	if(ch==4)
	{
		exit(0);
	}


	switch (ch)
	{
		case 1: //Sequential Write
			printf("\n\nEnter the number of threads(1/2) :\n");
			scanf("%d",&nthread);

			if(nthread==1 || nthread==2)
			{
```

```
printf("\nBYTE read for thread %d",nthread);
printf("\n\nSEQUENTIAL Write");
startTime=clock();
for(i=0;i<nthread;i++)
{
	pthread_create(&th[i],NULL,fileWriteByteSequential,NULL);
	pthread_join(th[i], NULL);
}
endTime = clock();
timeDiff = ((double) (endTime - startTime)) / CLOCKS_PER_SEC;
latency= (timeDiff*1000)/(double)(nthread);
printf("\nLatency : %f ms",latency/1000);
throughput=(BITERATIONS/(double)(latency*MEGABYTE));
printf("\nThroughtput:%f MB/s",throughput);

//Random Write
printf("\n\nRANDOM Write");
startTime=clock();
for(i=0;i<nthread;i++)
{
	pthread_create(&th[i],NULL,fileWriteByteRandom,NULL);
	pthread_join(th[i], NULL);
}
endTime = clock();
timeDiff = ((double) (endTime - startTime)) / CLOCKS_PER_SEC;
latency= (timeDiff*1000)/(double)(nthread);
printf("\nLatency : %f ms",latency/1000);
throughput=(BITERATIONS/(double)(latency*MEGABYTE));
printf("\nThroughtput:%f MB/s",throughput);


//Sequential Read
printf("\n\nSEQUENTIAL Read");
startTime=clock();
for(i=0;i<nthread;i++)
{
	pthread_create(&th[i],NULL,fileReadByteSequential,NULL);
	pthread_join(th[i], NULL);
}
endTime = clock();
timeDiff = ((double) (endTime - startTime)) / CLOCKS_PER_SEC;
latency= (timeDiff*1000)/(double)(nthread);
printf("\nLatency : %f ms",latency/1000);
throughput=(BITERATIONS/(double)(latency*MEGABYTE));
printf("\nThroughtput:%f MB/s",throughput);

//Random Read
printf("\n\nRANDOM Read");
startTime=clock();
```

```
		for(i=0;i<nthread;i++)
		{
			pthread_create(&th[i],NULL,fileReadByteRandom,NULL);
			pthread_join(th[i], NULL);
		}
		endTime = clock();
		timeDiff = ((double) (endTime - startTime)) / CLOCKS_PER_SEC;
		latency= (timeDiff*1000)/(double)(nthread);
		printf("\nLatency : %f ms",latency/1000);
		throughput=((BITERATIONS)/(double)(latency*MEGABYTE));
		printf("\nThroughtput:%f MB/s",throughput);

		break;
		}

		else{
			printf("\nInvalid thread\n");
			break;
		}

	case 2: //Sequential Write KiloByte

		printf("\n\nEnter the number of threads(1/2) :\n");
		scanf("%d",&nthread);

		if(nthread==1 || nthread==2)
		{

		printf("\n\nSEQUENTIAL Write KiloByte");
		startTime=clock();
		for(i=0;i<nthread;i++)
		{
			pthread_create(&th[i],NULL,fileWriteKiloByteSequential,NULL);
			pthread_join(th[i], NULL);
		}
		endTime = clock();
		timeDiff = ((double) (endTime - startTime)) / CLOCKS_PER_SEC;
		latency= (timeDiff*1000)/(double)(nthread);
		printf("\nLatency : %f ms",latency);
		throughput=(KBITERATIONS*1000)/(double)(latency*KILOBYTE);
		printf("\nThroughtput:%f MB/s",throughput);

		//Random Write KiloByte
		printf("\n\nRANDOM Write KiloByte");
		startTime=clock();
		for(i=0;i<nthread;i++)
		{
			pthread_create(&th[i],NULL,fileWriteKiloByteRandom,NULL);
			pthread_join(th[i], NULL);
		}
```

```
endTime = clock();
timeDiff = ((double) (endTime - startTime)) / CLOCKS_PER_SEC;
latency= (timeDiff*1000)/(double)(nthread);
printf("\nLatency : %f ms",latency);
throughput=(KBITERATIONS*1000)/(double)(latency*KILOBYTE);
printf("\nThroughtput:%f MB/s",throughput);

//Sequential Read KiloByte
printf("\n\nSEQUENTIAL Read KiloByte");
startTime=clock();
for(i=0;i<nthread;i++)
{
        pthread_create(&th[i],NULL,fileReadKiloByteSequential,NULL);
        pthread_join(th[i], NULL);
}
endTime = clock();
timeDiff = ((double) (endTime - startTime)) / CLOCKS_PER_SEC;
latency= (timeDiff*1000)/(double)(nthread);
printf("\nLatency : %f ms",latency);
throughput=(KBITERATIONS*1000)/(double)(latency*KILOBYTE);
printf("\nThroughtput:%f MB/s",throughput);

//Random Read KiloByte
printf("\n\nRandom Read KiloByte");
startTime=clock();
for(i=0;i<nthread;i++)
{
        pthread_create(&th[i],NULL,fileReadKiloByteRandom,NULL);
        pthread_join(th[i], NULL);
}
endTime = clock();
timeDiff = ((double) (endTime - startTime)) / CLOCKS_PER_SEC;
latency= (timeDiff*1000)/(double)(nthread);
printf("\nLatency : %f ms",latency);
throughput=(KBITERATIONS*1000)/(double)(latency*KILOBYTE);
printf("\nThroughtput:%f MB/s",throughput);
break;
}
else
{
        printf("\nInvalid thread\n");
        break;
}


case 3: //Sequential Write MegaByte
        printf("\n\nEnter the number of threads(1/2) :\n");
        scanf("%d",&nthread);

        if(nthread==1 || nthread==2)
```

```
{
printf("\n\nSEQUENTIAL Write MegaByte");
startTime=clock();
for(i=0;i<nthread;i++)
{
        pthread_create(&th[i],NULL,fileWriteMegaByteSequential,NULL);
        pthread_join(th[i], NULL);
}
endTime = clock();
timeDiff = ((double) (endTime - startTime)) / CLOCKS_PER_SEC;
latency= (timeDiff*1000)/(double)(nthread);
printf("\nLatency : %f ms",latency);
throughput=((MBITERATIONS*1000)/((double)(latency)));
printf("\nThroughtput:%f MB/s",throughput);

//Random Write MegaByte
printf("\n\nRANDOM Write MegaByte");
startTime=clock();
for(i=0;i<nthread;i++)
{
        pthread_create(&th[i],NULL,fileWriteMegaByteRandom,NULL);
        pthread_join(th[i], NULL);
}
endTime = clock();
timeDiff = ((double) (endTime - startTime)) / CLOCKS_PER_SEC;
latency= (timeDiff*1000)/(double)(nthread);
printf("\nLatency : %f ms",latency);
throughput=((MBITERATIONS*1000)/((double)(latency)));
printf("\nThroughtput:%f MB/s",throughput);

//Sequential Read MegaByte
printf("\n\nSequential Read MegaByte");
startTime=clock();
for(i=0;i<nthread;i++)
{
        pthread_create(&th[i],NULL,fileReadMegaByteSequential,NULL);
        pthread_join(th[i], NULL);
}
endTime = clock();
timeDiff = ((double) (endTime - startTime)) / CLOCKS_PER_SEC;
latency= (timeDiff*1000)/(double)(nthread);
printf("\nLatency : %f ms",latency);
throughput=((MBITERATIONS*1000)/((double)(latency)));
printf("\nThroughtput:%f MB/s",throughput);

//Random Read MegaByte
printf("\n\nRandom Read MegaByte");
startTime=clock();
for(i=0;i<nthread;i++)
{
```

```
                        pthread_create(&th[i],NULL,fileReadMegaByteRandom,NULL);
                        pthread_join(th[i], NULL);
                }
                endTime = clock();
                timeDiff = ((double) (endTime - startTime)) / CLOCKS_PER_SEC;
                latency= (timeDiff*1000)/(double)(nthread);
                printf("\nLatency : %f ms",latency);
                throughput=((MBITERATIONS*1000)/((double)(latency)));
                printf("\nThroughtput:%f MB/s",throughput);

                break;
                }
                else{
                printf("\nInvalid thread\n");
                break;
                }

        case 4: exit(0);
                break;
        default: printf("\nOOPSS...Please enter valid input");
    }
    };

}
```

Memory Benchmarking: (C)

MemoryBenchmark.c

```
#include<stdio.h>
#include<sys/time.h>
#include<string.h>
#include<stdlib.h>
#include<pthread.h>

#define BLOCK_SIZE 1
#define BLOCK_SIZE_KB 1024
#define BLOCK_SIZE_MB 1024*1024

#define BITERATIONS 100000000
#define KBITERATIONS 10000000
#define MBITERATIONS 100

void *block_Byte();
void *block_Byte_random();


void *block_Kbyte()
{
	int i;
	int len;
	double a=5;
	long k= 0;
	char *mem = malloc(1000*sizeof(*mem));
	for(k=0;k<1000;k++)
	{
		mem[k]='c';
	}
	char *mem_write=malloc(BLOCK_SIZE_KB*sizeof(*mem_write));

	for(i=0;i<KBITERATIONS;i++)
	{
		memcpy(mem_write,&mem,BLOCK_SIZE_KB);
		*(mem_write+i);
	}
}

void *block_KByte_random()
{
	int i,r;
	int len;
	double a=5;
	long k= 0;
	char *mem = malloc(1000*sizeof(*mem));
	for(k=0;k<1000;k++)
	{
		mem[k]='c';
	}
	char *mem_write=malloc(BLOCK_SIZE_KB*sizeof(*mem_write));
```

```
	for(i=0;i<KBITERATIONS;i++)
	{
		r = rand()%BLOCK_SIZE_KB;
		memcpy(mem_write + r,&mem +r,BLOCK_SIZE_KB);
	}
}

void *block_Mbyte()
{
	int i;
	int len;
	double a=5;
	long k= 0;
	char *mem = malloc(1000*sizeof(*mem));
	for(k=0;k<1000;k++)
	{
		mem[k]='c';
	}
	char *mem_write=malloc(BLOCK_SIZE_MB*sizeof(*mem_write));

	for(i=0;i<MBITERATIONS;i++)
	{
		memcpy(mem_write,&mem,BLOCK_SIZE_MB);
		*(mem_write+i);
	}
}

void *block_MByte_random()
{
	int i,r;
	int len;
	double a=5;
	long k= 0;
	char *mem = malloc(1000*sizeof(*mem));
	for(k=0;k<1000;k++)
	{
		mem[k]='c';
	}
	char *mem_write=malloc(BLOCK_SIZE_MB*sizeof(*mem_write));

	for(i=0;i<MBITERATIONS;i++)
	{
		r = rand()%MBITERATIONS;
		memcpy(mem_write + r,&mem +r,BLOCK_SIZE_MB);
	}
}

int main()
{

	clock_t start_t, end_t, total_t=0,start_t1, end_t1,total_t1=0;
	double latency,throughput,timeDiff;
	double lat;
	printf("\n.....Program to find Memory Benchmark.....");
```

```
pthread_t th[10];// array of threads
int i;
int ch,nthread;
while(1)
{
	printf("\n\nEnter the Block Size:\n1.BYTE\n2.KILOBYTE\n3.MEGABYTE\n4.Exit\n");

	scanf("%d",&ch);
	if(ch==4)
	{
		exit(0);
	}
	switch (ch)
	{
		case 1:
		printf("\nEnter the number of threads(1/2) :\n");
		scanf("%d",&nthread);

		//Sequential Memroy Read+Write
		if(nthread==1 || nthread==2)
		{

		printf("\nByte read+write for %d thread",nthread);
		printf("\n..Sequential Read+Write..\n");
		printf("-------------------------------------");

		start_t = clock();
		for(i=0;i<nthread;i++)
		{
			pthread_create(&th[i],NULL,block_Byte,NULL);
			pthread_join(th[i], NULL);
		}

		end_t=clock();
		timeDiff = ((double) (end_t - start_t)) / CLOCKS_PER_SEC;
		latency= (timeDiff*1000)/(double)(nthread);
		printf("\nLatency : %f ms",latency/1000);
		throughput=(BITERATIONS/(double)(latency*BLOCK_SIZE_MB));
		printf("\nThroughput:%f MB/s",throughput*1000);


		// Random Memroy Read+Write

		printf("\n\n..Random read+write..\n");
		printf("-------------------------------------");
		start_t=clock();
		for(i=0;i<nthread;i++)
		{
			pthread_create(&th[i],NULL,block_Byte_random,NULL);
			pthread_join(th[i], NULL);
		}
		end_t=clock();
		timeDiff = ((double) (end_t - start_t)) / CLOCKS_PER_SEC;
		latency= (timeDiff*1000)/(double)(nthread);
		printf("\nLatency : %f ms",latency/1000);
```

```
throughput=(BITERATIONS/(double)(latency*BLOCK_SIZE_MB));
printf("\nThroughput:%f MB/s",throughput*1000);
break;
}
else{
printf("\nInvalid thread\n");
break;
}

case 2:
printf("\nEnter the number of threads(1/2) :\n");
scanf("%d",&nthread);

//Sequential Memroy Read+Write for KILOBYTE block

if(nthread==1 || nthread==2)
{
printf("\n\nKiloByte read+write for %d thread ",nthread);
printf("\n..Sequential Read+Write..\n");
printf("-------------------------------------");
start_t = clock();
for(i=0;i<nthread;i++)
{
        pthread_create(&th[i],NULL,block_Kbyte,NULL);
        pthread_join(th[i], NULL);
}
end_t=clock();
timeDiff = ((double) (end_t - start_t)) / CLOCKS_PER_SEC;
latency= (timeDiff*1000)/(double)(nthread);
printf("\nLatency : %f ms",latency);
throughput=(KBITERATIONS/(double)(latency*BLOCK_SIZE_KB));
printf("\nThroughput:%f MB/s",throughput*1000);


//Random Memroy Read+Write for KILOBYTE block

printf("\n\n..Random read+write..\n");
printf("-------------------------------------");
start_t = clock();
for(i=0;i<nthread;i++)
{
        pthread_create(&th[i],NULL,block_KByte_random,NULL);
        pthread_join(th[i], NULL);
}
end_t=clock();
timeDiff = ((double) (end_t - start_t)) / CLOCKS_PER_SEC;
latency= (timeDiff*1000)/(double)(nthread);
printf("\nLatency : %f ms",latency);
throughput=(KBITERATIONS/(double)(latency*BLOCK_SIZE_KB));
printf("\nThroughput:%f MB/s",throughput*1000);
break;
}
else{
printf("\nInvalid thread\n");
break;
```

```
}

case 3:
printf("\nEnter the number of threads(1/2) :\n");
scanf("%d",&nthread);

//Sequential Memroy Read+Write

if(nthread==1 || nthread==2)
{
printf("\nMegaByte read+write for %d thread",nthread);
printf("\n..Sequential Read+Write..\n");
printf("-------------------------------------");
start_t = clock();
for(i=0;i<nthread;i++)
{
        pthread_create(&th[i],NULL,block_Mbyte,NULL);
        pthread_join(th[i], NULL);
}

end_t=clock();
timeDiff = ((double) (end_t - start_t)) / CLOCKS_PER_SEC;
latency= (timeDiff*1000)/(double)(nthread);
printf("\nLatency : %f ms",latency/1000);
throughput=(MBITERATIONS/(double)(latency));
printf("\nThroughput:%f MB/s",throughput*1000);


//Random Memory Read+Write

printf("\n\n..Random read+write..\n");
printf("-------------------------------------");
start_t1 = clock();
for(i=0;i<nthread;i++)
{
        pthread_create(&th[i],NULL,block_MByte_random,NULL);
        pthread_join(th[i], NULL);
}
end_t=clock();
timeDiff = ((double) (end_t - start_t)) / CLOCKS_PER_SEC;
latency= (timeDiff*1000)/(double)(nthread);
printf("\nLatency : %f ms",latency/1000);
throughput=(MBITERATIONS/(double)(latency));
printf("\nThroughput:%f MB/s",throughput*1000);

break;
}
else{
printf("\nInvalid thread\n");
break;
}

case 4: exit(0);
        break;
default: printf("\nPlease enter a valid option..\n");
```

```
                }
        }
}



void *block_Byte()
{
        int i;
        int len;
        double a=5;
        long k= 0;
        char *mem = malloc(sizeof(*mem));
        mem[0]='c';
        char *mem_write=malloc(sizeof(*mem_write));

        for(i=0;i<BITERATIONS;i++)
        {
                memcpy(mem_write,&mem,BLOCK_SIZE);
                *(mem_write+i);
        }
}

void *block_Byte_random()
{
        int i,r;
        int len;
        double a=5;
        long k= 0;

        char *mem = malloc(sizeof(*mem));
        mem[k]='c';

        char *mem_write=malloc(sizeof(*mem_write));

        for(i=0;i<BITERATIONS;i++)
        {
                r = rand()%BLOCK_SIZE;
                memcpy(mem_write + r,&mem +r,BLOCK_SIZE);
        }
}
```