**MULTICORE Architecture AND PROGRAMMING**

**eXERCISE PROGRAMS**

**Contents:**

|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Topic** | **Page Nos.** |
| **1** | **Pthread programs** | **2-14** |
| **2** | **Winthread programs** | **15-25** |
| **3** | **OpenMP programs** | **26-42** |
| **4** | **MPI programs** | **43-51** |

**1. Pthread Programs**

Compile: ***gcc pgm.c -lpthread***

**1. Program to create child threads**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUM\_THREADS 20

void \*PrintHello(void \*threadid)

{

long tid;

tid = (long)threadid;

printf("Hello World! It's me, thread #%ld!\n", tid);

pthread\_exit(NULL);

}

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

{

pthread\_t threads[NUM\_THREADS];

int rc;

long t;

for(t=0; t<NUM\_THREADS; t++)

{

printf("In main: creating thread %ld\n", t);

rc = pthread\_create(&threads[t], NULL, PrintHello, (void \*)t);

if (rc)

{

printf("ERROR; return code from pthread\_create() is %d\n", rc);

exit(-1);

}

}

pthread\_exit(NULL);

}

**2. Program to pass arguments to threads**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUM\_THREADS 4

struct thread\_data{

int thread\_id;

int sum;

char \*message;

};

struct thread\_data thread\_data\_array[NUM\_THREADS];

void \*PrintHello(void \*threadarg)

{

struct thread\_data \*my\_data;

int taskid, sum;

char\* hello\_msg;

my\_data = (struct thread\_data \*) threadarg;

taskid = my\_data->thread\_id;

sum = my\_data->sum;

hello\_msg = my\_data->message;

printf("id=%d\n",taskid);

printf("sum=%d\n",sum);

printf("msg=%s\n",hello\_msg);

}

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

{

//...

int t, rc;

pthread\_t threads[NUM\_THREADS];

for(t=0;t<NUM\_THREADS;t++)

{

thread\_data\_array[t].thread\_id = t;

thread\_data\_array[t].sum = 0;

thread\_data\_array[t].message = "hello world";

printf("thread %d created\n",t);

rc = pthread\_create(&threads[t], NULL, PrintHello,(void \*) &thread\_data\_array[t]);

printf("thread %u created\n",threads[t]);

}

//...

return 0;

}

**3. program to make main thread to wait for child threads to exit**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void \*PrintThreads (void \*);

#define NUM\_THREADS 5

int main()

{

int i, ret;

pthread\_t thdHandle [NUM\_THREADS];

int thdNum[NUM\_THREADS];

for (i=0 ; i< NUM\_THREADS; i++)

thdNum[i] = i;

for (i=0 ; i< NUM\_THREADS; i++)

{

ret = pthread\_create ( &thdHandle[i], NULL, PrintThreads, (void \*) &thdNum[i] );

if (ret==0)

printf("thread %d launched successfully\n",i);

}

// join all the threads

for (i=0 ; i< NUM\_THREADS; i++)

pthread\_join (thdHandle[i], NULL);

printf("end of main...\n");

return 0;

}

void \*PrintThreads ( void \*num)

{

int i,j;

for (i=0; i<3; i++)

{

printf("thread number is %d \n",\*((int\*)num));

if(i<2)

scanf("%d",&j); //make this thread simply wait; so that others may execute

}

return (NULL);

}

**4. Program to illustrate the use of mutexes**

//Count the number of lines and spaces in a text file.

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#define NUMTHREADS 5

FILE \*fd;

int glc = 0, gsc=0;

//declare and initialize the mutexes

pthread\_mutex\_t amutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t bmutex = PTHREAD\_MUTEX\_INITIALIZER;

void \*countlines(void \*num)

{

int bDone = 0 ;

char inLine[100];

int spc, lnc,j;

int n = \*((int\*)num);

while (!bDone)

{

// synchronize the access to the input file

pthread\_mutex\_lock(&amutex);

if (fgets(inLine,100,fd)==NULL) bDone=1;

pthread\_mutex\_unlock(&amutex);

if (!bDone)

{

spc=0; lnc=0;

for( j=0; inLine[j]!='\0'; j++)

{

if(inLine[j]==' ')

spc++;

if (inLine[j]=='\n')

lnc++;

}

//synchronize the access to gsc and lsc

pthread\_mutex\_lock(&bmutex);

gsc+= spc;

glc+= lnc;

pthread\_mutex\_unlock(&bmutex);

}

}

return;

}

int main()

{

pthread\_t thd\_handle[NUMTHREADS];

int mynum[20];

int i,ret;

fd = fopen("infile.txt","r");

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

{

mynum[i]= i;

ret=pthread\_create(&thd\_handle[i],NULL,countlines,&mynum[i]);

}

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

pthread\_join(thd\_handle[i],NULL);

fclose(fd);

printf("Total number of spaces= %d\t and lines=%d\n", gsc, glc);

return 0;

}

**5.Program to compute dot product- using mutex**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define NTHREADS 4

#define ARRAYSIZE 100

#define ITERATIONS ARRAYSIZE / NTHREADS

double sum=0.0, a[ARRAYSIZE], b[ARRAYSIZE];

pthread\_mutex\_t sum\_mutex;

void \*do\_work(void \*tnum)

{

int i, start, \*mytnum, end;

double mysum=0.0;

/\* Initialize local sum \*/

mytnum = (int \*) tnum;

start = (\*mytnum \* ITERATIONS);

end = start + ITERATIONS;

if ((\*mytnum)== NTHREADS-1) end= ARRAYSIZE;

printf ("Thread %d doing iterations %d to %d\n",\*mytnum,start,end-1);

for (i=start; i < end ; i++)

{

mysum = mysum + a[i]\*b[i];

}

/\* Lock the mutex and update the global sum and exit \*/

pthread\_mutex\_lock (&sum\_mutex);

sum = sum + mysum;

pthread\_mutex\_unlock (&sum\_mutex);

pthread\_exit(NULL);

}

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

{

int i, start, thrnum[NTHREADS];

double serial\_sum=0;

pthread\_t tid[NTHREADS];

// Initialize the arrays

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

{

a[i] = i\*1.0;

b[i] = i\*2.0;

serial\_sum = serial\_sum + (a[i]\*b[i]);

}

/\* Create threads \*/

pthread\_mutex\_init(&sum\_mutex, NULL);

for (i=0; i<NTHREADS; i++) {

thrnum[i] = i;

pthread\_create(&tid[i], NULL, do\_work, (void \*) &thrnum[i]);

}

/\* Wait for all threads to complete and print the global sum \*/

for (i=0; i<NTHREADS; i++) {

pthread\_join(tid[i], NULL);

}

printf ("Parallel- Sum= %14.2f \n", sum);

printf("Serial Check- Sum= %14.2f\n",serial\_sum);

/\*Release the mutex resources \*/

pthread\_mutex\_destroy(&sum\_mutex);

pthread\_exit (NULL);

}

**6. Program to compute dot product - using barrier**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define NTHREADS 4

#define ARRAYSIZE 100

#define ITERATIONS ARRAYSIZE / NTHREADS

pthread\_barrier\_t barrier;

pthread\_mutex\_t sum\_mutex;

double sum=0.0, a[ARRAYSIZE], b[ARRAYSIZE], locsum[NTHREADS];

void \*do\_work(void \*tnum)

{

int i, start, \*mytnum, end;

/\* Initialize local sum \*/

mytnum = (int \*) tnum;

start = (\*mytnum \* ITERATIONS);

end = start + ITERATIONS;

if ((\*mytnum)== NTHREADS-1) end= ARRAYSIZE;

printf ("Thread %d doing iterations %d to %d\n",\*mytnum,start,end-1);

for (i=start; i < end ; i++)

{

locsum[\*mytnum] = locsum[\*mytnum] + a[i]\*b[i];

}

// pthread\_barrier\_wait(&barrier);

/\* Lock the mutex and update the global sum and exit \*/

if (\*mytnum==0)

{

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

sum = sum + locsum[i];

}

pthread\_exit(NULL);

}

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

{

int i, start, thrnum[NTHREADS];

double serial\_sum=0;

pthread\_t tid[NTHREADS];

pthread\_barrier\_init(&barrier,NULL, NTHREADS);

// Initialize the arrays

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

{

a[i] = i\*1.0;

b[i] = i\*2.0;

serial\_sum = serial\_sum + (a[i]\*b[i]);

}

/\* Create threads \*/

pthread\_mutex\_init(&sum\_mutex, NULL);

for (i=0; i<NTHREADS; i++) {

locsum[i]=0.0;

thrnum[i] = i;

pthread\_create(&tid[i], NULL, do\_work, (void \*) &thrnum[i]);

}

/\* Wait for all threads to complete and print the global sum \*/

for (i=0; i<NTHREADS; i++) {

pthread\_join(tid[i], NULL);

}

printf ("Parallel- Sum= %14.2f \n", sum);

printf("Serial (Check)- Sum= %14.2f\n",serial\_sum);

/\*Release the mutex resources \*/

pthread\_mutex\_destroy(&sum\_mutex);

pthread\_exit (NULL);

}

**7. Program to illustrate the usage of monitors in POSIX threads**

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#define NUMTHREADS 2

pthread\_mutex\_t mutex=PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t cv=PTHREAD\_COND\_INITIALIZER;

int a,b,flag=0;

void \*compute(void \*num)

{

int \*n;

n= (int \*) num;

pthread\_mutex\_lock(&mutex);

while (flag==0)

pthread\_cond\_wait(&cv, &mutex);

pthread\_mutex\_unlock(&mutex);

printf("Thread%d is signalled\n",\*n);

if (\*n==0) printf("The sum is %d\n",a+b);

else if(\*n==1) printf("The difference is %d\n",a-b);

}

int main()

{

pthread\_t tid[2];

int thdno[2]={0, 1};

int i,ret;

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

{pthread\_create(&tid[i], NULL, compute, (void \*) &thdno[i]);

printf("Thread%d is created\n",i);

}

pthread\_mutex\_lock(&mutex);

printf("Enter two numbers ");

scanf("%d%d",&a,&b);

flag=1;

pthread\_cond\_broadcast(&cv);

pthread\_mutex\_unlock(&mutex);

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

pthread\_join(tid[i], NULL);

return 0;

}

**8. Program to illustrate the usage of monitors in POSIX threads**

**//Counts lines and spaces in a text file**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUMTHREADS 2

char inline1[100],inline2[100];

FILE \*infile;

int glc = 0, gsc=0, done=0;

pthread\_mutex\_t addmutex = PTHREAD\_MUTEX\_INITIALIZER;

typedef struct

{

pthread\_mutex\_t mutex;

pthread\_cond\_t cv;

int data;

} flag;

flag flag1= { PTHREAD\_MUTEX\_INITIALIZER,

PTHREAD\_COND\_INITIALIZER,

0

};

flag flag2= { PTHREAD\_MUTEX\_INITIALIZER,

PTHREAD\_COND\_INITIALIZER,

0

};

//Thread function-1 to Read and process data from buffer1

void \*count1(void \* arg)

{

int spc=0, lnc=0,j;

while (1)

{

if (feof(infile)) break;

//Read from buffer1

pthread\_mutex\_lock(&flag1.mutex);

while (flag1.data==0 && !feof(infile))

pthread\_cond\_wait(&flag1.cv, &flag1.mutex);

if(flag1.data)

for( j=0; inline1[j]!='\0'; j++)

{

if(inline1[j]==' ')

spc++;

if (inline1[j]=='\n')

lnc++;

}

flag1.data=0;

pthread\_mutex\_unlock(&flag1.mutex);

}

//Synchronize the access to gsc and lsc

pthread\_mutex\_lock(&addmutex);

gsc+= spc;

glc+= lnc;

pthread\_mutex\_unlock(&addmutex);

pthread\_exit (NULL);

}

//Thread function-2 to Read and process data from buffer2

void \*count2( void \* arg)

{

int spc=0, lnc=0,j;

while (1)

{

if (feof(infile)) break;

//Read from buffer2

pthread\_mutex\_lock(&flag2.mutex);

while (flag2.data==0 && !feof(infile))

pthread\_cond\_wait(&flag2.cv, &flag2.mutex);

if(flag2.data)

for( j=0; inline2[j]!='\0'; j++)

{

if(inline2[j]==' ')

spc++;

if (inline2[j]=='\n')

lnc++;

}

flag2.data=0;

pthread\_mutex\_unlock(&flag2.mutex);

} //endwhile

//Synchronize the access to gsc and lsc

pthread\_mutex\_lock(&addmutex);

gsc+= spc;

glc+= lnc;

pthread\_mutex\_unlock(&addmutex);

pthread\_exit (NULL);

}

int main()

{

char in[10];

pthread\_t tid[2];

int thdno[2]={0, 1};

int i;

infile = fopen("infile.txt", "r+b");

if (infile == NULL)

{

printf("error opening file %s\n",in);

return(-1);

}

pthread\_create(&tid[0], NULL,count1, NULL);

pthread\_create(&tid[1], NULL,count2, NULL);

//Main Thread to write data into buffer1 and buffer2

while (!feof(infile))

{

//Write into buffer1

pthread\_mutex\_lock(&flag1.mutex);

if(flag1.data==0)

if (fgets(inline1,100,infile)!=NULL)

flag1.data=1;

//Signal thread1 whether or not flag1.data=1

pthread\_cond\_signal(&flag1.cv);

pthread\_mutex\_unlock(&flag1.mutex);

//Write into buffer2

pthread\_mutex\_lock(&flag2.mutex);

if(flag2.data==0)

if (fgets(inline2,100,infile)!=NULL )

flag2.data=1;

// Signal thread2 whether or not flag2.data=1

pthread\_cond\_signal(&flag2.cv);

pthread\_mutex\_unlock(&flag2.mutex);

}//endwhile

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

pthread\_join(tid[i],NULL);

printf("Total number of spaces= %d\t and lines=%d\n", gsc, glc);

fclose(infile);

pthread\_exit (NULL);

}

**9. Program to illustrate the usage of semaphores in POSIX threads**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define NUMTHREADS 4

sem\_t sem;

void \*worker(void \* n)

{

int \* num;

num= (int\*) n;

sem\_wait(&sem);

printf("Thread %d is inside the Critical Section\n",\*num);

sleep(1);

sem\_post(&sem);

printf("Thread %d is outside the Critical Section\n",\*num);

}

int main()

{

pthread\_t tid[NUMTHREADS];

int tnum[NUMTHREADS];

int i;

sem\_init(&sem,0,2);

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

{ tnum[i]=i;

pthread\_create(&tid[i], NULL, worker, (void \*) &tnum[i]);

}

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

pthread\_join(tid[i],NULL);

sem\_destroy(&sem);

}

**Program 10: Count number of lines and spaces using semaphore to synchronize**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

char buf[100];

FILE \*infile;

int spc = 0, lnc=0, done=0, data=0;

sem\_t s;

void \*count(void \* arg)

{

int j;

while (1)

{

sem\_wait(&s);

if (data)

{

//Process the data from the buffer

for( j=0; buf[j]!='\0'; j++)

{

if(buf[j]==' ')

spc++;

if (buf[j]=='\n')

lnc++;

}

data=0;

if (!data) done=1;

}

if(done && feof(infile) )

break;

}

pthread\_exit (NULL);

}

int main()

{

pthread\_t tid;

sem\_init(&s,0,0);

infile = fopen("infile.txt", "r+b");

if (infile == NULL)

{

printf("error opening the input file\n");

return(-1);

}

pthread\_create(&tid, NULL,count, NULL);

//Read data from file into the buffer

while (!feof(infile))

{

if (data==0)

{ if (fgets(buf,100,infile)!=NULL)

data=1;

}

sem\_post(&s);

}

pthread\_join(tid,NULL);

printf("Total number of spaces= %d\t and lines=%d\n", spc, lnc);

fclose(infile);

sem\_destroy(&s);

pthread\_exit (NULL);

}

**2. WINDOWS THREAD PROGRAMS**

**Note: These programs can be executed in visual studio or codeblocks IDE**

// r1.cpp : thread ceation : main thread may not wait for the

// child thread to execute

#include"stdafx.h"//for visual studio only; not for code bloccks

#include<stdio.h>

#include<conio.h>

#include<windows.h>

DWORD WINAPI helloFunc(LPVOID arg )

{

**printf(**"Hello Thread\n");

return **0;**

}

**//\_tmain() in visual studi; main() in codeblocks**

int **\_tmain(**int **argc, \_TCHAR\* argv[])** {

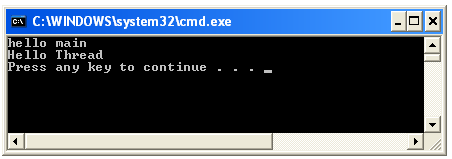
**printf(**"hello main\n");

**HANDLE hThread = CreateThread(NULL, 0, helloFunc, NULL, 0, NULL );**

return **0;**

}

Output 1



Output 2



// r2.cpp : creating a thread : main thread waits for the

// child thread to execute. (not a right method)

#include"stdafx.h"

#include<stdio.h>

#include<windows.h>

BOOL threadDone = FALSE ;

DWORD WINAPI helloFunc(LPVOID arg )

{

**printf(**"Hello Thread\n");

**threadDone = TRUE ;**

return **0;**

}

int **\_tmain(**int **argc, \_TCHAR\* argv[])**

{

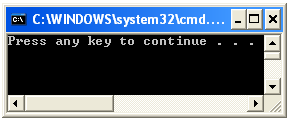
**HANDLE hThread = CreateThread(NULL, 0, helloFunc,NULL, 0, NULL );**

while **(!threadDone);**

return **0;**

}

Output :



// r3.cpp : Main thread waits for child threads to complete //execution ( the right method)

#include"stdafx.h"

#include<stdio.h>

#include<windows.h>

constint **numThreads = 4;**

DWORD WINAPI helloFunc(LPVOID pArg )

{

int **mynum = \*((**int\*)pArg);

**printf(**"Hello Thread %d\n",mynum);

return **0;**

}

int **\_tmain(**int **argc, \_TCHAR\* argv[])**

{ int **tNum[4];**

**HANDLE hThread[numThreads];**

for **(**int **i = 0; i < numThreads; i++)**

**{**

**tNum[i] = i;**

**hThread[i] = CreateThread(NULL, 0, helloFunc, &tNum[i], 0, NULL);**

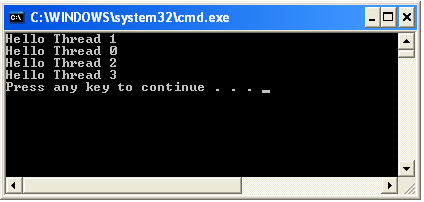
**}**

**WaitForMultipleObjects(numThreads, hThread, TRUE, INFINITE);**

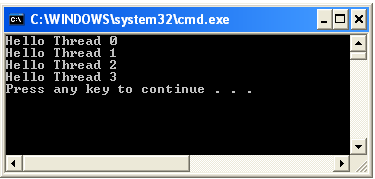
return **0;**

}

Output1



Output2



// r4.cpp : passing parameters to thread through a structure

// Pass i by value (cv) and by reference (addr)

#include<alloc.h>

#include"stdafx.h"

#include<stdio.h>

#include<windows.h>

constint **numThreads = 10;**

struct **ab**

**{**

int **\*adr;**

int **cv;**

**};**

DWORD WINAPI helloFunc(LPVOID pArg )

{

struct **ab \*r = (**struct **ab\*)pArg;**

int **\*q= r->adr;**

int **mynum=(r->cv);**

int **cur\_val=\*q;**

**printf(**"Hello Thread %d... value of i in main=%d\n",mynum, cur\_val);

return **0;**

}

int **\_tmain(**int **argc, \_TCHAR\* argv[])**

{

struct **ab\* p =** newstruct **ab;**

**HANDLE hThread[numThreads];**

for **(**int **i = 0; i < numThreads; i++)**

**{ printf(**"In main..now i=%d\n",i);

**p->adr=&i;**

**p->cv=i;**

**hThread[i] = CreateThread(NULL, 0, helloFunc, p, 0, NULL );**

**printf(**"thread %d called\n\n",i);

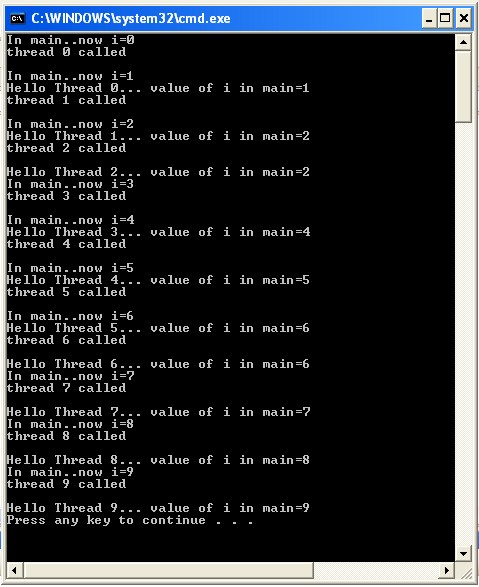
**}**

**WaitForMultipleObjects(numThreads, hThread, TRUE, INFINITE);**

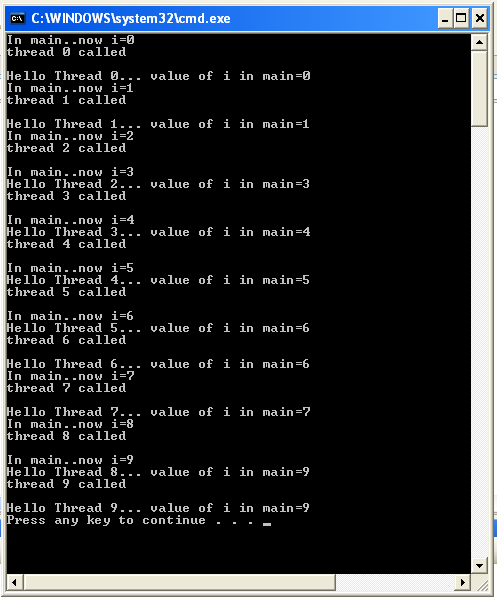
return **0;**

}

Output 1 : observe the thread execution sequence and parameter values



Output 2



// semaphore.cpp : pgm to illustrate use of semaphores. this pgm //counts the no. of lines and spaces in a given input file

#include <stdio.h>

#include <windows.h>

#include <conio.h>

#define NUMTHREADS 50

HANDLE hSem1, hSem2;

FILE \*fd;

int glc = 0, gsc=0;

DWORD WINAPI Countlines(LPVOID arg)

{

bool bDone = FALSE ;

char inLine[100];

int spc, lnc;

int t\_spc=0, t\_lnc=0;

int num = \*((int\*)arg);

while (!bDone)

{

WaitForSingleObject(hSem1, INFINITE); // access to input

if (fgets(inLine,100,fd)==NULL) bDone=1;

ReleaseSemaphore(hSem1, 1, NULL);

if (!bDone)

{

spc=0; lnc=0;

for(int j=0; inLine[j]!='\0'; j++)

{

if(inLine[j]==' ')

spc++;

if (inLine[j]=='\n')

lnc++;

} //endfor

WaitForSingleObject(hSem2, INFINITE); // update global count

gsc+= spc;

Sleep(1);

glc+= lnc;

ReleaseSemaphore(hSem2, 1, NULL);

} //endif

} //endwhile

return(0);

}

int main()

{

HANDLE hThread[NUMTHREADS];

int mynum[100];

hSem1 = CreateSemaphore(NULL, 1, 1, NULL); // Binary semaphore

hSem2 = CreateSemaphore(NULL, 1, 1, NULL); // Binary semaphore

if(hSem1==NULL) printf("error1..\n");

if(hSem2==NULL) printf("error2..\n");

fd = fopen("infile.txt","r"); // Open file for read

for (int i = 0; i < NUMTHREADS; i++)

{

mynum[i]= i;

hThread[i] = CreateThread(NULL,0,Countlines,&mynum[i],0,NULL);

}

WaitForMultipleObjects(NUMTHREADS, hThread, TRUE, INFINITE);

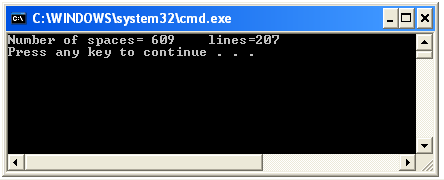
fclose(fd);

printf("Number of spaces= %d\t lines=%d\n", gsc, glc);

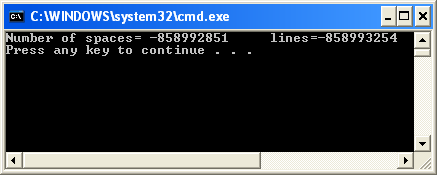
return 0;

}

Output 1 – when semaphores are used



Output 1 – when semaphores are not used



// event.cpp : thread events

#include"stdafx.h"

#include<stdio.h>

#include<windows.h>

#include<stdlib.h>

HANDLE hObj[2]; // 0 is event, 1 is thread

DWORD WINAPI threadFunc(LPVOID arg)

{

**BOOL bFound =0;** //bigFind();

int **a[20], i, key;**

**printf(**"the list..\n");

for(i=0;i<20;i++)

**{a[i]=i;**

**printf(**"%d ",a[i]);

**}**

**printf(**"\nenter the key..");

**scanf(**"%d",&key);

for(i=0;i<20;i++)

**{**

if(a[i]==key)

**bFound=1;**

**}**

if **(bFound)**

**{**

**SetEvent(hObj[0]);** // signal data was found

//bigFound() ;

**}**

if **(!bFound)**

**printf(**"Key Not Found\n");

//moreBigStuff() ;

return **0;**

}

int **\_tmain(**int **argc, \_TCHAR\* argv[])**

{

**hObj[0] = CreateEvent(NULL, FALSE, FALSE, NULL);** // create a thread

**hObj[1] = CreateThread(NULL,0,threadFunc,NULL,0,NULL);** // create an event

/\* Do some other work while thread executes search \*/

**DWORD waitRet = WaitForMultipleObjects(2, hObj, FALSE, 2000);**

switch(waitRet)

**{**

case **WAIT\_OBJECT\_0:** // event signaled

**printf(**"found it!... event signalled\n");

**WaitForSingleObject(hObj[1], INFINITE) ;**

// fall thru

break;

case **WAIT\_OBJECT\_0+1:** // thread signaled

**printf(**"not found.. thread terminates\n");

break **;**

case **WAIT\_TIMEOUT:** // thread timed out

**printf(**"thread timed out\n");

break **;**

default:

**printf(**"wait error: ret %u\n", waitRet);

break **;**

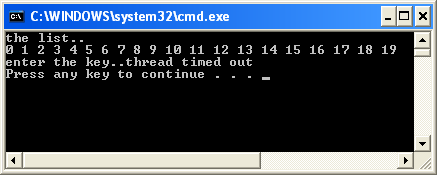
**}**

//getch();

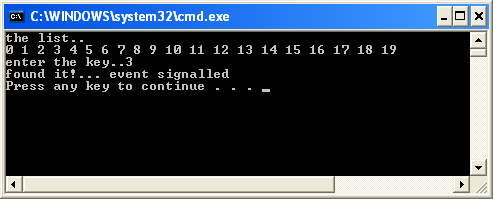
return **0;**

}

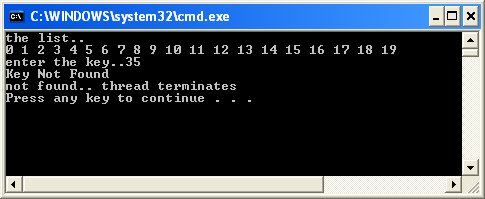
Outut 1



Output 2



Output 3



**3. OpenMP Programs**

**Command to compile OpenMP program in ubuntu:**

**gcc prg\_name.c –fopenmp**

**Command to run:**

**./a.out**

**// Program-1 to create a parallel region**

#include <stdio.h>

#include<omp.h>

int main(void)

{

int a=10, b=20, sum;

#pragma omp parallel //Beginning of parallel region

{

sum=a+b;

printf("The Sum=%d\n",sum);

}

//End of parallel region

return 0;

}

**// Program 2- Illustrate the use of global and local**

**// variables in a parallel** **region**

#include <stdio.h>

#include<omp.h>

int main(void)

{

int a=10;

#pragma omp parallel

{

int product;

int tid;

tid=omp\_get\_thread\_num();

if(tid==0)

{product=a\*5;

printf("From thread-%d : The product=%d\n",tid,product);

}

else

{ product=a\*6;

printf("From thread-%d: The product=%d\n",tid,product);

}

}

return 0;

}

**// Program 3- To illustrate the usage of thread-number,**

**// number-of-threads and number-of-processors functions**

#include <stdio.h>

#include <stdlib.h>

#include<omp.h>

int main(void)

{

printf("There are totally %d processors in the system\n", omp\_get\_num\_procs());

//omp\_set\_num\_threads(2);

#pragma omp parallel

{

int tid;

tid=omp\_get\_thread\_num();

if (tid==0)

printf("There are totally %d threads in the parallel region\n", omp\_get\_num\_threads());

printf("Hello from thread %d\n",tid);

}

return 0;

}

**//Program 4: To illustrate the effect of using num\_threads clause**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main()

{

#pragma omp parallel num\_threads(2)

{if(omp\_get\_thread\_num()==0)

printf("The number of threads when num\_threads(2) clause is used=%d\n\n",omp\_get\_num\_threads());

}

#pragma omp parallel

{if(omp\_get\_thread\_num()==0)

printf("The number of threads without num\_threads() clause=%d\n",omp\_get\_num\_threads());

}

return 0;

}

**//Program 5: To illustrate concept of if nested parallel regions**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main()

{

omp\_set\_nested(-1);

#pragma omp parallel

{if(omp\_get\_thread\_num()==0)

printf("The number of threads in the outer parallel region=%d\n\n",omp\_get\_num\_threads());

printf("Hello from outer thread %d\n\n", omp\_get\_thread\_num());

#pragma omp parallel

{if(omp\_get\_thread\_num()==0)

printf("The number of threads in the inner parallel region=%d\n\n",omp\_get\_num\_threads());

printf("Hello from inner thread %d\n\n", omp\_get\_thread\_num());

}

}

return 0;

}

**//Program 6: To enable and disable dynamic parallelism**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main()

{

//If dynamic adjustment is enabled then the requested number of

// threads is the maximum number of threads that may execute

//the parallel region. In our system max no. of threads=2

omp\_set\_dynamic(1);

printf("Dynamic parallelism is Enabled\n");

#pragma omp parallel num\_threads(4)

{if(omp\_get\_thread\_num()==0)

printf("The number of threads in the parallel region 1=%d\n\n",omp\_get\_num\_threads());

}

//If dynamic adjustment is disabled,

// then the requested number of threads will execute the parallel region.

omp\_set\_dynamic(0);

printf("Dynamic parallelism is Disabled\n");

#pragma omp parallel num\_threads(4)

{if(omp\_get\_thread\_num()==0)

printf("The number of threads in the parallel region 2=%d\n\n",omp\_get\_num\_threads());

}

return 0;

}

**//program 7: Conditional Compilation**

#include <stdio.h>

# ifdef \_OPENMP

#include<omp.h>

# endif // \_OPENMP

int main(void)

{

int par=1;

# ifdef \_OPENMP

#pragma omp parallel if(par)

# endif

printf("Hello, world.\n\n");

return 0;

}

**//program 8 : Perform C=A+B and D=A\*B using parallel for.**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 30

int main()

{

int nthreads;

unsigned int i; //i as unsigned int works

float a[N],b[N],c[N],d[N];

/\*Initialize arrays \*/

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

a[i]=b[i]=i\*1.0;

omp\_set\_num\_threads(2);

#pragma omp parallel

{/\*Parallel region begins \*/

int tid, print\_flag=1;

tid=omp\_get\_thread\_num();

if (tid==0)

printf("Number of threads =%d\n",omp\_get\_num\_threads());

#pragma omp for

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

{

c[i]= a[i]+b[i];

d[i]= a[i]\*b[i];

if(print\_flag)

{

printf("Thread-%d begins at iteration %d\n",tid,i);

print\_flag=0;

}

}

} /\* End of the parallel region \*/

}

**//Program 8: Add two arrays; ie C=A+B.**

**// Observe execution time using OpenMP lib function omp\_get\_wtime()**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <stdint.h>

#define CHUNKSIZE 1000

#define N 100000

int main()

{

int nthreads,tid, i, chunk;

float a[N],b[N], c[N],d[N];

double start, end, time\_used;

chunk=CHUNKSIZE;

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

a[i]=b[i]=i\*1.0;

omp\_set\_num\_threads(2);

start=(double)omp\_get\_wtime();

#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i, tid)

{

tid=omp\_get\_thread\_num();

if(tid==0)

printf("Number of threads =%d\n",omp\_get\_num\_threads());

//#pragma omp for schedule(static, chunk)

#pragma omp for schedule(runtime)

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

{

c[i]= a[i]+b[i];

d[i]= a[i]\*b[i];

}

}

end= (double) omp\_get\_wtime();

printf("The execution time = %10.10lf seconds\n", (end - start));

}

**//program 9 :Illustrate the usage of single construct**

//**Compute dot product- code uses single construct**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 30

int main()

{

int i,sum=0;

float a[N],b[N],c[N];

#pragma omp parallel num\_threads(2)

{

#pragma omp single nowait

{

printf("Number of threads =%d\n",omp\_get\_num\_threads());

}

#pragma omp single

{

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

a[i]=b[i]= i\*1.0;

}

#pragma omp for

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

{

c[i]= a[i]\*b[i];

}

#pragma omp single

{

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

sum=sum+c[i];

}

}

printf("Dot Product=%d\n",sum);

return 0;

}

**//Program 10: Illustrate the usage of threadprivate variables**

#include <stdio.h>

#include <omp.h>

int tp1,p;

float tp2;

#pragma omp threadprivate(tp1, tp2)

int main ()

{

int tid;

omp\_set\_dynamic(0); // To enable threadprivate variables

tp1=5;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Parallel Region-1\n");

#pragma omp parallel private(tid,p) copyin(tp1)

{

tid = omp\_get\_thread\_num();

tp1+=tid;

tp2+=tid;

p=tp1+tp2;

printf("Thread %d: tp1=%d, tp2=%0.1f, p=%d\n",tid,tp1,tp2,p);

} /\* end of parallel section \*/

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Serial Region \n");

tid = omp\_get\_thread\_num();

printf("Thread %d: tp1=%d, tp2=%0.1f, p=%d\n",tid,tp1,tp2,p);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Parallel Region-2\n");

#pragma omp parallel private(tid)

{

tid = omp\_get\_thread\_num();

printf("Thread %d: tp1=%d, tp2=%0.1f\n",tid,tp1,tp2);

} /\* end of parallel section \*/

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

return 0;

}

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parallel Region-1

Thread 0: tp1=5, tp2=0.0, p=5

Thread 1: tp1=6, tp2=1.0, p=7

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Serial Region

Thread 0: tp1=5, tp2=0.0, p=0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parallel Region-2

Thread 1: tp1=6, tp2=1.0

Thread 0: tp1=5, tp2=0.0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

**//program 11 : Demonstrate the properties of private variables**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int g=100;

int sum(int, int);

void main()

{

int a=10, b=5,c;

printf("Before the parallel region: a=%d b=%d g=%d\n",a,b,g);

omp\_set\_num\_threads(2);

printf("Within the parallel region\n");

#pragma omp parallel private(a,b,g)

{

a=omp\_get\_thread\_num();

b=10;

c=sum(a,b);

printf("Thread-%d a=%d b=%d c=%d g=%d\n",a,a,b,c,g);

}

printf("After the parallel region: a=%d b=%d g=%d\n",a,b,g);

}

int sum(int x, int y)

{

return x+y;

}

/\*

Before the parallel region: a=10 b=5 g=100

Within the parallel region

Thread-0 a=0 b=10 c=10

Thread-1 a=1 b=10 c=11

After the parallel region: a=10 b=5 g=100

\*/

**//Program 12: To illustrate the effects of firstprivate clause.**

//Program 17: To illustrate the effects of firstprivate clause.

#include <stdio.h>

#include <omp.h>

void main()

{

int x, y, tid;

x = 100;

y = 200;

printf("In the parallel region\n");

#pragma omp parallel firstprivate(x) private(tid)

{

tid=omp\_get\_thread\_num();

x = x + tid;

#pragma omp critical

{y=y+5;}

#pragma omp barrier

//A thread waits here till all threads reach this point

printf("Thread-%d x=%d y=%d\n",tid,x,y);

#pragma omp sections firstprivate(y)

//firstprivate(tid)is not possible here.

{

#pragma omp section

{

y=y+1;

printf("\nIn worksharing section-1: Thread-%d x=%d y=%d\n",tid,x,y);

}

#pragma omp section

{

y=y+2;

printf("\nIn worksharing section-2: Thread-%d x=%d y=%d\n",tid,x,y);

}

}// end of sections pragma

} // end of parallel pragma

printf("\nAfter the parallel region x=%d y=%d \n",x,y);

}

/\*

In the parallel region

Thread-0 x=100 y=210

Thread-1 x=101 y=210

In worksharing section-1: Thread-0 x=100 y=211

In worksharing section-2: Thread-1 x=101 y=212

After the parallel region x=100 y=210

\*/

**//Program 13: To illustrate the lastprivate clause.**

#include <stdio.h>

#include <omp.h>

void main()

{

int lp=0,i,tid;

printf("In the parallel region\n");

#pragma omp parallel for firstprivate(lp), lastprivate(lp) private(tid)

for(i=0;i<10;i++)

{

tid=omp\_get\_thread\_num();

lp = lp+i;

printf("thread %d i=%d lp=%d\n",tid,i,lp);

}

printf("\nAfter the parallel region lp=%d \n",lp);

}

/\*

In the parallel region

thread 0 i=0 lp=0

thread 1 i=5 lp=5

thread 0 i=1 lp=1

thread 0 i=2 lp=3

thread 0 i=3 lp=6

thread 1 i=6 lp=11

thread 1 i=7 lp=18

thread 1 i=8 lp=26

thread 1 i=9 lp=35

thread 0 i=4 lp=10

After the parallel region lp=35

\*/

**//Program 14: To illustrate properties of shared variables.**

#include <stdio.h>

#include <omp.h>

void main()

{

int s1=0, s2=0, s3=0,i;

omp\_set\_num\_threads(2);

#pragma omp parallel shared(s1,s2,s3)

{

#pragma omp critical

{s1=s1+5;}

#pragma omp for

for(i=0;i<10;i++)

{

#pragma omp critical

{s2=s2+i;}

}

#pragma omp sections

{

#pragma omp section

{

#pragma omp critical

{s3=s3+5;}

}

#pragma omp section

{

#pragma omp critical

{s3=s3+10;}

}

}

}

printf("Outside the parallel region\n");

printf("s1=%d s2=%d s3=%d\n",s1,s2,s3);

}

/\*when shared clause is not used:

Outside the parallel region

s1=10 s2=45 s3=15

when used:

Outside the parallel region

s1=10 s2=45 s3=15

\*/

**//Program 15: To illustrate the effects of using default clause**

//Program 21: To illustrate the effects of using default clause.

#include <stdio.h>

#include <omp.h>

int a[10],b,c=0,d=0;

#pragma omp threadprivate(b)

void main()

{

int x=0,i=0;

const int y = 10;

#pragma omp parallel default(none) private(x) shared(a,d)

{

int tid = omp\_get\_thread\_num();

//tid is private; declared inside the parallel region.

a[tid] = y;/OK - a is specified in shared clause, tid is private.

// - y is const type-qualified.

b = x+5; //OK - b is specified in threadprivate.

// - x is specified in private clause.

c = i; //Error- cannot reference i or c here;

//both are not specified in the enclosing parallel construct.

#pragma omp for private(c) //private(c) is OK here

for (i=0; i<10 ; i++)

{

a[i] = i; //OK - can reference i ; loop index variable.

} // - a is a is specified in shared clause.

#pragma omp for firstprivate(x,c,d)

//Error- firstpivate(x) is not OK here as x is private in the

//outer context. firstprivate(c) not OK here because c is not

//specified in the enclosing parallel construct.

//firstprivate(d)is OK here.

for (i=0; i<10 ; i++)

{

a[i] = c; // referencing i is OK - loop index variable.

// referencing a is OK - specified in shared clause.

} // can't reference c; not specified in enclosing

// parallel construct.

c = i; // Error - cannot reference c and i here;

} // both not specified in enclosing parallel construct.

}

**//Program 16: Compute dot product using reduction.**

#include <stdio.h>

#include <omp.h>

#define N 30

int main()

{

int i;

float a[N],b[N],sum=0.0;

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

a[i]=b[i]= i\*1.0;

#pragma omp parallel for reduction(+:sum)

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

sum = sum + a[i] \* b[i];

printf("Dot Product=%.1f\n",sum);

return 0;

}

**//Program 17: Compute dot product using reduction.**

**//The reduction operator is used inside the function.**

#include <stdio.h>

#include <omp.h>

#define N 30

float add(float,float,float);

int main()

{

int i;

float a[N],b[N],sum=0.0;

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

a[i]=b[i]= i\*1.0;

#pragma omp parallel for reduction(+:sum)

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

sum = add(sum,a[i],b[i]); //operator can be inside the function

printf("Dot Product=%.1f\n",sum);

return 0;

}

float add(float s, float n1, float n2)

{

return s+n1\*n2;

}

**//Program 18: Illustrate the usage of copyin clause**

#include <stdio.h>

#include <omp.h>

int tp1;

float tp2;

#pragma omp threadprivate(tp1,tp2)

int main ()

{

int tid;

omp\_set\_dynamic(0); // To enable threadprivate variables

omp\_set\_num\_threads(4);

tp1=5; tp2=5.5;

printf("Serial Region \n");

tid = omp\_get\_thread\_num();

printf("Master thread- thread-id %d: tp1=%d,

tp2=%0.1f\n",tid,tp1,tp2);

printf("-------------------------------------------------\n");

printf("Parallel Region-1: Without copyin\n");

#pragma omp parallel private(tid)

{

tid = omp\_get\_thread\_num();

printf("Thread-id %d: tp1=%d, tp2=%0.1f\n",tid,tp1,tp2);

}

printf("-------------------------------------------------\n");

printf("Serial Region \n");

tid = omp\_get\_thread\_num();

printf("Master thread - thread-id %d: tp1=%d,

tp2=%0.1f\n",tid,tp1,tp2);

printf("-------------------------------------------------\n");

printf("Parallel Region-2: With copyin\n");

#pragma omp parallel private(tid) copyin(tp1,tp2)

{

tid = omp\_get\_thread\_num();

printf("Thread-id %d: tp1=%d, tp2=%0.1f\n",tid,tp1,tp2);

}

printf("-------------------------------------------------\n");

return 0;

}

Ouput:

Serial Region

Master thread- thread-id 0: tp1=5, tp2=5.5

-------------------------------------------------

Parallel Region-1: Without copyin

Thread-id 0: tp1=5, tp2=5.5

Thread-id 2: tp1=0, tp2=0.0

Thread-id 1: tp1=0, tp2=0.0

Thread-id 3: tp1=0, tp2=0.0

-------------------------------------------------

Serial Region

Master thread - thread-id 0: tp1=5, tp2=5.5

-------------------------------------------------

Parallel Region-2: With copyin

Thread-id 2: tp1=5, tp2=5.5

Thread-id 1: tp1=5, tp2=5.5

Thread-id 0: tp1=5, tp2=5.5

Thread-id 3: tp1=5, tp2=5.5

-------------------------------------------------

**//Program 19: Illustrate the effects of copyprivate clause.**

#include <stdio.h>

#include <omp.h>

float x=2.0;

#pragma omp threadprivate(x)

int main()

{

float a = 1.0;

omp\_set\_num\_threads(4);

#pragma omp parallel private(a) //*a* must not be shared here.

{

int tid=omp\_get\_thread\_num();

a=tid;

x+=tid;

printf("Before updating- Thread-%d: a=%.1f,

x=%.1f\n",tid,a,x);

#pragma omp barrier

#pragma omp single copyprivate(a, x) //private(a) or

{ //firstprivate(a)

a++; //not allowed here.

x++;

printf("\nThread-%d updated a and x: a=%.1f,

x=%.1f\n\n",tid,a,x);

}

printf("After updating- Thread-%d: a=%.1f,

x=%.1f\n",tid,a,x);

}

}

Output:

Before updating- Thread-0: a=0.0, x=2.0

Before updating- Thread-2: a=2.0, x=4.0

Before updating- Thread-3: a=3.0, x=5.0

Before updating- Thread-1: a=1.0, x=3.0

Thread-3 updated a and x: a=4.0, x=6.0

After updating- Thread-2: a=4.0, x=6.0

After updating- Thread-3: a=4.0, x=6.0

After updating- Thread-0: a=4.0, x=6.0

After updating- Thread-1: a=4.0, x=6.0

**//Program 20: Illustrate functioning of master construct.**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main()

{

int tid;

omp\_set\_num\_threads(4);

#pragma omp parallel

{

tid=omp\_get\_thread\_num();

#pragma omp master private(tid)

{

printf("Inside the master construct: Thread-id %d\n",tid);

}

printf("Outside the master construct: Thread-id %d\n",tid);

}

}

Output:

Inside the master construct: Thread-id 0

Outside the master construct: Thread-id 1

Outside the master construct: Thread-id 2

Outside the master construct: Thread-id 3

Outside the master construct: Thread-id 0

**//Program 21: Illustrate the usage of critical sections**

#include <stdio.h>

#include <omp.h>

main()

{

long int i;

double sum=0.0, diff=499500.0;

omp\_set\_num\_threads(4);

#pragma omp parallel for shared(sum)

for(i=0;i<1000;i++)

{

#pragma omp critical

{

sum=sum+i;

diff=diff-i;

}

}

printf("The sum=%.1lf difference=%.1lf\n",sum,diff);

}

Output:

The sum=499500.0 difference=0.0

**//Program 22: Illustrate the usage of named critical sections**

#include <stdio.h>

#include <omp.h>

main()

{

int x,y;

x=y=0;

omp\_set\_num\_threads(10);

#pragma omp parallel shared(x,y)

{

#pragma omp critical (CSA)

{x = x + 1;

}

#pragma omp critical (CSB)

{y = y + 1;

}

}

printf("x=%d y=%d\n",x,y);

}

Output :

x=10 y=10

**//Program 23: Illustrate the use of barrier**

#include <stdio.h>

#include <omp.h>

int main()

{

int tid;

float a=5,b=10,x=4,y=5,sum=0.0,p1,p2;

#pragma omp parallel private(tid)

{

tid=omp\_get\_thread\_num();

if(tid==0)

{

//Do some computation

p1=a\*x;

}

if(tid==1)

{

//Do some computation

p2=b\*y;

}

//Wait for the other thread to finish computation

#pragma omp barrier

#pragma omp single

sum=p1+p2;

}

printf("Sum=%.1f\n",sum);

}

Output:

Sum=70.0

**//Program 24: TO illustrate the usage of atomic construct.**

#include <stdio.h>

#include <omp.h>

void main()

{

int x,y;

x=y=0;

omp\_set\_num\_threads(50);

#pragma omp parallel shared(x,y)

{

#pragma omp atomic

x = x + 1;

#pragma omp atomic

y = y + 1;

}

printf("x=%d y=%d\n",x,y);

}

Output:

x=50 y=50

**//Program 25: Illustrate the working of ordered clause and**

**//ordered directive.**

#include <stdio.h>

#include <omp.h>

#define N 10

float a[N], b[N], c[N];

int main( )

{

int i;

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

{

a[i]=i;

b[i]=i\*2.0;

}

#pragma omp parallel

{

#pragma omp single

printf("With ordered construct \n");

#pragma omp for ordered

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

{

c[i]=a[i]+b[i];

#pragma omp ordered

printf("%d\n",i);

}

#pragma omp single

printf("\nWithout ordered construct \n");

#pragma omp for

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

{

c[i]=a[i]+b[i];

printf("%d\n",i);

}

}

}

Output:

With ordered construct

0 1 2 3 4 5 6 7 8 9

Without ordered construct

5 0 6 1 7 2 3 4 8 9

**//Program 26: Illustrate the use of simple lock.**

#include <stdio.h>

#include <omp.h>

#define N 1000

int main()

{

int i,p;

int a[N],b[N];

long int sum=0;

omp\_lock\_t L;

omp\_init\_lock(&L);

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

a[i]=b[i]= i\*1.0;

omp\_set\_num\_threads(4);

#pragma omp parallel for shared(sum, L) private(p)

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

{

p=a[i] \* b[i];

omp\_set\_lock(&L);

sum = sum +p;

omp\_unset\_lock(&L);

}

printf("Dot Product=%ld\n",sum);

omp\_destroy\_lock(&L);

return 0;

}

**//Program 27: Illustrate the functioning of nested lock.**

#include <stdio.h>

#include <omp.h>

int a=0;

omp\_nest\_lock\_t L;

void step2\_compute(void)

{

int i;

omp\_set\_nest\_lock(&L);

for(i=0;i<1000;i++) //Do some computation

a += 2;

omp\_unset\_nest\_lock(&L);

}

void step1\_compute(void)

{

int i;

omp\_set\_nest\_lock(&L);

for(i=0;i<1000;i++) //Do some computation

a += 1;

step2\_compute();

omp\_unset\_nest\_lock(&L);

}

void main()

{

omp\_init\_nest\_lock(&L);

omp\_set\_num\_threads(2);

#pragma omp parallel sections

{

#pragma omp section

step1\_compute();

#pragma omp section

step2\_compute();

}

printf("x=%d\n",a);

omp\_destroy\_nest\_lock(&L);

}

**4. MPI Programs**

**// command to compile mpi program :**

* + **mpicc mpi\_code.c –o a.out**

**// command to run mpi program on two machines:**

* + **mpirun -np 2 a.out**

**// Program 1. To illustrate Blocking Message Passing Routines // All worker processes send a greeting message to**

**// master process.**

#include <stdio.h>

#include "mpi.h"

#include <string.h>

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

{

    int my\_rank, p;   // process rank and number of processes

    int source, dest; // rank of sender and receiving process

    int tag = 0;       // tag for messages

    char mesg[100];   // storage for message

    MPI\_Status status;// stores status for MPI\_Recv statements

  MPI\_Init(&argc, &argv);

  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);

  MPI\_Comm\_size(MPI\_COMM\_WORLD, &p);

   if (my\_rank!=0)

{

        sprintf(mesg, "Greetings from %d!", my\_rank);

// stores into character array

        dest = 0;

        MPI\_Send(mesg, strlen(mesg)+1, MPI\_CHAR, dest, tag,

MPI\_COMM\_WORLD);

}// sends string to process 0

else

{

        for(source = 1; source < p; source++){

        MPI\_Recv(message, 100, MPI\_CHAR, source, tag,

MPI\_COMM\_WORLD, &status);

// recv from each process

        printf("%s\n", message);    }

    MPI\_Finalize(); // shuts down MPI

}

**//Program 2. Master process send a message to all worker**

**//process using Blocking Message Passing Routines.**

#include <stdio.h>

#include "mpi.h"

#include <string.h>

#define BUFFER\_SIZE 32

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

{

int MyRank, Numprocs, Destination;

int tag = 0;

int Root = 0, dest;

char Message[BUFFER\_SIZE];

MPI\_Init(&argc,&argv);

MPI\_Status status;

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&MyRank);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&Numprocs);

/\* print host name, and send message from process with rank 0 to all other processes \*/

if(MyRank == 0) {

strcpy(Message, "Hello India");

for (dest=1; dest<Numprocs; dest++)

MPI\_Send(Message, BUFFER\_SIZE, MPI\_CHAR, dest,tag,MPI\_COMM\_WORLD);

}

else {

system("hostname"); // execute the OS commannd - *hostname*

MPI\_Recv(Message, BUFFER\_SIZE, MPI\_CHAR, Root, tag,MPI\_COMM\_WORLD, &status);

printf("\n%s in process with rank %d from Process with rank %d\n", Message,MyRank,Root);

}

MPI\_Finalize();

}

**//Program 3. MPI program to pass a message around a ring.**

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

{

// Initialize the MPI environment

MPI\_Init(NULL, NULL);

// Find out rank, size

int world\_rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

int world\_size;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

  int token;

// Receive from the lower process and send to the

// higher process. Take care of the special case when you

// are the first process to prevent deadlock.

if (world\_rank != 0)

{

MPI\_Recv(&token, 1, MPI\_INT, world\_rank - 1, 0,

MPI\_COMM\_WORLD,MPI\_STATUS\_IGNORE);

printf("Process %d received token %d from process %d\n“,

world\_rank, token, world\_rank - 1);

}

else

{

// Set the token's value if you are process 0

token = -1;

}

MPI\_Send(&token, 1, MPI\_INT, (world\_rank + 1) % world\_size,

0, MPI\_COMM\_WORLD);

// Now process 0 can receive from the last process. This

// makes sure that at least one MPI\_Send is initialized

// before all MPI\_Recvs (again, to prevent deadlock)

if (world\_rank == 0)

{

MPI\_Recv(&token, 1, MPI\_INT, world\_size - 1, 0,

MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Process %d received token %d from process %d\n",

world\_rank, token,world\_size - 1);

}

MPI\_Finalize();

}

**//Program 4**. **MPI program to compute the value of**

#include "mpi.h"

#include <math.h>

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

{

int done = 0, n, myid, numprocs, i, rc;

double PI25DT = 3.141592653589793238462643;

double mypi, pi, h, sum, x, a;

MPI\_Init(&argc,&argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&numprocs);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&myid);

while (!done)

{

if (myid == 0)

{

printf("Enter the number of intervals: (0 quits) ");

scanf("%d",&n);

}

MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (n == 0) break;

h = 1.0 / (double) n;

sum = 0.0;

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

{

x = h \* ((double)i - 0.5);

sum += 4.0 / (1.0 + x\*x);

}

mypi = h \* sum;

MPI\_Reduce(&mypi, &pi, 1, MPI\_DOUBLE, MPI\_SUM,

0,MPI\_COMM\_WORLD);

if (myid == 0)

printf("pi is approximately %.16f, Error is %.16f\n", pi,

fabs(pi - PI25DT));

}

MPI\_Finalize();

return 0;

}

**//Program 5**. To find sum of 'n' integers on 'p' processors using

// point-to-point communication libraries calls

#include <stdio.h>

#include "mpi.h"

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

{

int iproc;

int MyRank, Numprocs, Root = 0;

int value, sum = 0;

int Source, Source\_tag;

int Destination, Destination\_tag;

MPI\_Status status;

MPI\_Init(&argc,&argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&Numprocs);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&MyRank);

if(MyRank == Root){

for(iproc = 1 ; iproc < Numprocs ; iproc++)

{

Source = iproc;

Source\_tag = 0;

MPI\_Recv(&value, 1, MPI\_INT, Source, Source\_tag,

MPI\_COMM\_WORLD, &status);

sum = sum + value;

}

printf("MyRank = %d, SUM = %d\n", MyRank, sum);

}

Else

{

Destination = 0;

Destination\_tag = 0;

MPI\_Send(&MyRank, 1, MPI\_INT, Destination, Destination\_tag,

MPI\_COMM\_WORLD);

}

MPI\_Finalize();

}

**//Program 6. Read in an integer from a file (in.dat) and broadcast it to**

**// all processes**

#include <stdio.h>

#include "mpi.h"

#define ROOT 0

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

{

int size, rank, x=0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

if (rank == ROOT)

{

FILE \*fp = fopen("in.dat","r");

fscanf(fp,"%d",&x); fclose(fp);

}

printf("(before) my rank = %d, x = %d\n", rank, x);

MPI\_Bcast(&x, 1, MPI\_INT, ROOT, MPI\_COMM\_WORLD);

printf("(after) my rank = %d, x = %d\n", rank, x);

MPI\_Finalize();

return 0;

}

// **Program 7.** Program to illustrate isend/ireceive non-blocking communication

//library calls

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <math.h>

int main(argc,argv)

int argc;

char \*argv[];

{

int myid, numprocs;

int tag,source,destination,count;

int buffer;

MPI\_Status status;

MPI\_Request request;

MPI\_Init(&argc,&argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&numprocs);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&myid);

tag=1234;

source=0;

destination=1;

count=1;

request=MPI\_REQUEST\_NULL;

if(myid == source)

{

buffer=5678;

PI\_Isend(&buffer,count,MPI\_INT,destination,tag,MPI\_COMM\_WORLD,&request);

}

if(myid == destination)

{

MPI\_Irecv(&buffer,count,MPI\_INT,source,tag,MPI\_COMM\_WORLD,&request);

}

MPI\_Wait(&request,&status);

if(myid == source)

{

printf("processor %d sent %d\n",myid,buffer);

}

if(myid == destination)

{

printf("processor %d got %d\n",myid,buffer);

}

MPI\_Finalize();

}

/\* **Program 8. Program to illustrate the usage of MPI\_Scatter and MPI\_Reduce Each processor gets different data from the root processor by way of mpi\_scatter. The data is summed and then sent back to the root processor using MPI\_Reduce. The root processor then prints the global sum. \*/**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

/\* globals \*/

int numnodes,myid,mpi\_err;

#define mpi\_root 0

/\* end globals \*/

void init\_it(int \*argc, char \*\*\*argv);

void init\_it(int \*argc, char \*\*\*argv)

{

mpi\_err = MPI\_Init(argc,argv);

mpi\_err = MPI\_Comm\_size( MPI\_COMM\_WORLD, &numnodes );

mpi\_err = MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid);

}

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

int \*myray,\*send\_ray,\*back\_ray;

int count;

int size,mysize,i,k,j,total,gtotal;

init\_it(&argc,&argv);

/\* each processor will get count elements from the root \*/

count=4;

myray=(int\*)malloc(count\*sizeof(int));

/\* create the data to be sent on the root \*/

if(myid == mpi\_root){

size=count\*numnodes;

send\_ray=(int\*)malloc(size\*sizeof(int));

back\_ray=(int\*)malloc(numnodes\*sizeof(int));

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

send\_ray[i]=i;

}

/\* send different data to each processor \*/

mpi\_err = MPI\_Scatter( send\_ray, count, MPI\_INT, myray, count,

MPI\_INT, mpi\_root, MPI\_COMM\_WORLD);

/\* each processor does a local sum \*/

total=0;

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

total=total+myray[i];

printf("myid= %d total= %d\n ",myid,total);

/\* send the local sums back to the root \*/

mpi\_err = MPI\_Reduce(&total, &gtotal, 1, MPI\_INT, MPI\_SUM, mpi\_root,

MPI\_COMM\_WORLD);

/\* the root prints the global sum \*/

if(myid == mpi\_root){

printf("results from all processors= %d \n ",gtotal);

}

mpi\_err = MPI\_Finalize();

}

**//Program 9. Matrix-Matrix Multiplication By Dividing the work among all the**

**//available processes and Process with Rank 0 will print the result.**

#include "mpi.h"

#include <stdio.h>

#include <stdlib.h>

#define NoofRowsA 4 /\* numer of rows in matrix A \*/

#define NoofColsA 4 /\* numer of columns in matrix A \*/

#define NoofColsB 4 /\* numer of columns in matrix B \*/

#define MASTER 0 /\* MyRank of first task \*/

#define FROM\_MASTER 1 /\* setting a message type (Tag)\*/

#define FROM\_WORKER 2 /\* setting a message type (Tag)\*/

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

{

int numprocs, /\* numer of Processes in COMM\_WORLD \*/

MyRank, /\* a Process identifier \*/

numworkers, /\* numer of worker Processes \*/

source, /\* source \*/

dest, /\* destination \*/

mtype, /\* message type \*/

rows, /\* rows of matrix A sent to each worker \*/

averow, extra, offset, /\* used to determine rows sent to each worker \*/

i, j, k, rc,irow,icol; /\* misc \*

double A[NoofRowsA][NoofColsA], /\* matrix A to be multiplied \*/

B[NoofColsA][NoofColsB], /\* matrix B to e multiplied \*/

C[NoofRowsA][NoofColsB]; /\* result matrix C \*/

double starttime,endtime; /\*Parameters for getting the time required

for operation\*/

MPI\_Status status;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MPI Initialization\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

MPI\_Init(&argc,&argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&MyRank);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&numprocs);

if (numprocs < 2 )

{

printf("Need at least two MPI tasks. Quitting...\n");

MPI\_Abort(MPI\_COMM\_WORLD, rc);

exit(1);

}

numworkers = numprocs-1;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* master task \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (MyRank == MASTER)

{

printf("\n \*\*\*\*\*\*\*\*\*\*\*\*\*OUTPUT FOR THE PROGRAM\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

printf("\nmpi has started with %d tasks.\n",numprocs);

printf("Initializing Matrices...\n");

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Initializing the contents of Matrix A\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

for (j=0; j<NoofColsA; j++)

A[i][j]= i+j;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Initializing the contents of Matrix B\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

for (j=0; j<NoofColsB; j++)

B[i][j]= i\*j;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Send matrix data to the worker tasks\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

averow = NoofRowsA/numworkers;

extra = NoofRowsA%numworkers;

offset = 0;

mtype = FROM\_MASTER;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Getting the starting time for operation\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

starttime=MPI\_Wtime();

for (dest=1; dest<=numworkers; dest++)

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*Distributing the Rows among the No. Of Processes\*\*\*\*\*\*\*\*/

rows = (dest <= extra) ? averow+1 : averow;

printf("Sending %d rows to Rank %d offset=%d\n",rows,dest,offset);

MPI\_Send(&offset, 1, MPI\_INT, dest, mtype, MPI\_COMM\_WORLD);

MPI\_Send(&rows, 1, MPI\_INT, dest, mtype, MPI\_COMM\_WORLD);

MPI\_Send(&A[offset][0], rows\*NoofColsA, MPI\_DOUBLE, dest, mtype,

MPI\_COMM\_WORLD);

MPI\_Send(&B, NoofColsA\*NoofColsB, MPI\_DOUBLE, dest, mtype,

MPI\_COMM\_WORLD);

offset = offset + rows;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Receive results from worker tasks \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

mtype = FROM\_WORKER;

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

{

source = i;

MPI\_Recv(&offset, 1, MPI\_INT, source, mtype, MPI\_COMM\_WORLD,

&status);

MPI\_Recv(&rows, 1, MPI\_INT, source, mtype, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&C[offset][0], rows\*NoofColsB, MPI\_DOUBLE, source, mtype,

MPI\_COMM\_WORLD, &status);

printf("Received results from task %d\n",source);

/\*\*\*\*\*Calculating the End time at which the operation is compleated\*\*\*\*\*/

endtime=MPI\_Wtime();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Printing the results \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Final Result Matrix:\n");

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

{

printf("\n");

for (j=0; j<NoofColsB; j++)

printf("%6.2f ", C[i][j]);

}

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf ("Done.\n");

/\*\*\*\*\*\*\*\*\*\*\*Printing the total time required for the operation\*\*\*\*\*\*\*\*\*\*/

printf("\n Total time for operation is %.4f\n\n",endtime-starttime);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* worker task \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (MyRank > MASTER)

{

mtype = FROM\_MASTER;

MPI\_Recv(&offset, 1, MPI\_INT, MASTER, mtype, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&rows, 1, MPI\_INT, MASTER, mtype, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&A, rows\*NoofColsA, MPI\_DOUBLE, MASTER, mtype, MPI\_COMM\_WORLD,

&status);

MPI\_Recv(&B, NoofColsA\*NoofColsB, MPI\_DOUBLE, MASTER, mtype,

MPI\_COMM\_WORLD, &status);

/\*\*\*\*\*\*\*\*\*\*\*Worker Tasks performing the Matrix Multiplication\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

for (k=0; k<NoofColsB; k++)

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

{

C[i][k] = 0.0;

for (j=0; j<NoofColsA; j++)

C[i][k] = C[i][k] + A[i][j] \*B[j][k];

}

mtype = FROM\_WORKER;

MPI\_Send(&offset, 1, MPI\_INT, MASTER, mtype, MPI\_COMM\_WORLD);

MPI\_Send(&rows, 1, MPI\_INT, MASTER, mtype, MPI\_COMM\_WORLD);

MPI\_Send(&C, rows\*NoofColsB, MPI\_DOUBLE, MASTER, mtype,

MPI\_COMM\_WORLD);

}

MPI\_Finalize();

}