Lab 3

Tolaz Hewa

500590503

CPS590-01

Lab 3: Thread Scheduling

Lab 3: Thread Scheduling This lab creates a NUM\_THREADS (say 5) which all compute the adding from 1 to N (say N=10000) concurrently. It is useful to learn about the POSIX API to OS threads and to determine if threading will speed up computations.

POSIX Scheduling Policy:

SCHED\_FIFO: First In-First Out scheduling SCHED\_FIFO can only be used with static priorities higher than 0, which means that when a SCHED\_FIFO processes becomes runnable, it will always immediately preempt any currently running SCHED\_OTHER process.

SCHED\_RR: Round Robin scheduling SCHED\_RR is a simple enhancement of SCHED\_FIFO. Everything described above for SCHED\_FIFO also applies to SCHED\_RR, except that each process is only allowed to run for a maximum time quantum.

SCHED\_OTHER: Default Linux time-sharing scheduling SCHED\_OTHER can only be used at static priority 0. SCHED\_OTHER is the standard Linux time-sharing scheduler that is intended for all processes that do not require special static priority real-time mechanisms.

We set all threads to be run to completion at high priority so that we can determine whether the hardware provides speed-up by mapping threads onto multiple cores and/or SMT (Symmetric Multi-threading) on each core. Note that POSIX priorities are such that the highest priority thread has a large priority number.

Write a program to check and display:

• the currently used scheduling policy

• default attributes of thread

• current thread scope

• set new thread scope

• set FIFO policy with maximum priority

• check and display new attributes

• set new FIFO scheduling policy for all NUM\_THREADS

• calculate and display the execution time for all threads

• calculate and display the execution time for sequential operation for the thread function (same function without thread).

Is threading speed up the computation? Why?

A: Threading did speed up the process because all the calculations were being done concurrently. The threads increase the efficiency of the CPU and decrease running time because thread allow for all the work to be done asynchronously.

CODE:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/time.h>

#include <pthread.h>

#define NUM\_THREADS 5

#define N 10000000

void get\_info();

void \*blank(void \*arg);

void \*adder(void \*arg);

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

{

int i;

int policy;

time\_t startS, endS;

long startMS, endMS;

long diff\_in\_sec;

struct sched\_param param;

pthread\_attr\_t attr;

pthread\_t curr;

struct timeval tv;

pthread\_attr\_init(&attr);

printf("Before adjustments to scheduling policy:\n");

curr = pthread\_self();

get\_info(&curr);

pthread\_t threads[NUM\_THREADS];

int \*arr = (int\*)calloc(sizeof(int), NUM\_THREADS);

pthread\_getschedparam(pthread\_self(), &policy, &param);

pthread\_attr\_setschedpolicy(&attr, SCHED\_FIFO);

param.sched\_priority = 99;

pthread\_attr\_setschedparam(&attr, &param);

pthread\_create(&threads[0], &attr, blank , NULL);

get\_info(&threads[0]);

pthread\_join(threads[0], NULL);

gettimeofday(&tv, 0);

startMS = tv.tv\_usec;

startS = tv.tv\_sec;

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

{

arr[i] = i+1;

pthread\_create(&threads[i], &attr, adder, (void\*)&arr[i]);

}

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

{

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

}

gettimeofday(&tv, 0);

endMS = tv.tv\_usec;

endS = tv.tv\_sec;

diff\_in\_sec = endS - startS;

printf("\n\nElapsed time FOR THREADED TEST=%ld microseconds\n\n", (endMS - startMS) + (diff\_in\_sec \* 1000000));

gettimeofday(&tv, 0);

startMS = tv.tv\_usec;

startS = tv.tv\_sec;

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

{

adder(&arr[i]);

}

gettimeofday(&tv, 0);

endMS = tv.tv\_usec;

endS = tv.tv\_sec;

diff\_in\_sec = endS - startS;

printf("\n\nElapsed time for SEQUENTIAL TEST=%ld microseconds\n\n", (endMS - startMS) + (diff\_in\_sec \* 1000000));

free(arr);

return 0;

}

void \*blank(void \*arg)

{

return NULL;

}

void \*adder(void \*arg)

{

printf("%d: begin\n", \*((int\*)arg));

int i, sum;

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

{

sum += i;

}

printf("%d: Counter: %d\n", \*((int\*)arg), i);

printf("%d: done\n", \*((int\*)arg));

return NULL;

}

void get\_info(pthread\_t \*t)

{

pthread\_attr\_t attr;

int detach;

int scope;

int inherit;

int policy;

size\_t guard;

void \*base;

size\_t stack\_size;

struct sched\_param param;

pthread\_attr\_init(&attr);

pthread\_getschedparam(\*t, &policy, &param);

switch (policy)

{

case SCHED\_OTHER:

printf ("Pthread Policy is SCHED\_OTHER\n");

break;

case SCHED\_RR:

printf ("Pthread Policy is SCHED\_PR\n");

break;

case SCHED\_FIFO:

printf ("Pthread Policy is SCHED\_FIFO\n");

break;

case -1:

perror ("sched\_getscheduler returns error");

break;

default:

fprintf (stderr, "Unknown policy!\n");

}

pthread\_attr\_getdetachstate(&attr, &detach);

switch (detach)

{

case PTHREAD\_CREATE\_JOINABLE:

printf ("\tDetach state\t\t= PTHREAD\_CREATE\_JOINABLE\n");

break;

case PTHREAD\_CREATE\_DETACHED:

printf ("\tDetach state\t\t= PTHREAD\_CREATE\_DETACHED\n");

break;

case -1:

perror ("\tattr\_getdeatchstate returns error");

break;

default:

fprintf (stderr, "\tUnknown detach!\n");

}

pthread\_attr\_getscope(&attr, &scope);

switch (scope)

{

case PTHREAD\_SCOPE\_SYSTEM:

printf ("\tScope\t\t\t\t= PTHREAD\_SCOPE\_SYSTEM\n");

break;

case PTHREAD\_SCOPE\_PROCESS:

printf ("\tScope\t\t\t\t= PTHREAD\_SCOPE\_PROCESS\n");

break;

case -1:

perror ("\tattr\_getscope returns error");

break;

default:

fprintf (stderr, "\tUnknown scope!\n");

}

pthread\_attr\_getinheritsched(&attr, &inherit);

switch (inherit)

{

case PTHREAD\_INHERIT\_SCHED:

printf ("\tInherit schedule\t= PTHREAD\_INHERIT\_SCHED\n");

break;

case PTHREAD\_EXPLICIT\_SCHED:

printf ("\tInherit schedule\t= PTHREAD\_EXPLICIT\_SCHED\n");

break;

case -1:

perror ("\tattr\_getinheritsched returns error");

break;

default:

fprintf (stderr, "\tUnknown inherit scheduler!\n");

}

printf("\tScheduling Priority\t= %d\n", param.sched\_priority);

pthread\_attr\_getguardsize(&attr, &guard);

printf("\tGuard size\t\t\t= %d size\n", (int)guard);

pthread\_attr\_getstackaddr(&attr, &base);

printf("\tStack Address\t\t= %p\n", base);

pthread\_attr\_getstacksize(&attr, &stack\_size);

printf("\tStack Size\t\t\t= %lu bytes\n", stack\_size);

}

**INPUT/OUTPUT**

Before adjustments to scheduling policy:

Pthread Policy is SCHED\_OTHER

Detach state = PTHREAD\_CREATE\_JOINABLE

Scope = PTHREAD\_SCOPE\_SYSTEM

Inherit schedule = PTHREAD\_INHERIT\_SCHED

Scheduling Priority = 31

Guard size = 4096 size

Stack Address = 0x0

Stack Size = 524288 bytes

Pthread Policy is SCHED\_FIFO

Detach state = PTHREAD\_CREATE\_JOINABLE

Scope = PTHREAD\_SCOPE\_SYSTEM

Inherit schedule = PTHREAD\_INHERIT\_SCHED

Scheduling Priority = 99

Guard size = 4096 size

Stack Address = 0x0

Stack Size = 524288 bytes

1: begin

2: begin

3: begin

4: begin

2: Counter: 10000000

2: done

3: Counter: 10000000

3: done

4: Counter: 10000000

4: done

1: Counter: 10000000

1: done

5: begin

5: Counter: 10000000

5: done

Elapsed time FOR THREADED TEST=71926 microseconds

1: begin

1: Counter: 10000000

1: done

2: begin

2: Counter: 10000000

2: done

3: begin

3: Counter: 10000000

3: done

4: begin

4: Counter: 10000000

4: done

5: begin

5: Counter: 10000000

5: done

Elapsed time for SEQUENTIAL TEST=157957 microseconds