Nathan Bender

CS 283

H5 – Concurrency

**CSAPP3 Problem 12.16**

In this problem, I created N threads, where N is the number provided as a command line argument to my program. Each thread is created using Pthread\_create, calling a function called thread that prints the string “Hello World”. After each thread is created, I then loop over the thread IDs and join them when the main program, waiting for their execution to complete.

#include "csapp.h"

#include <stdio.h>

void \*thread(void \*vargp);

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

{

if (argc != 2) {

printf("Usage: ./%s <# of threads>\n", argv[0]);

exit(0);

}

int n = atoi(argv[1]);

pthread\_t tids[n];

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

{

Pthread\_create(&tids[i], NULL, thread, NULL);

}

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

{

Pthread\_join(tids[i], NULL);

}

exit(0);

}

void \*thread(void \*vargp)

{

printf("Hello World!\n");

return NULL;

}

**CSAPP3 Problem 12.17**

1. The problem in this program is that the program executes the statement "exit(0)"

which causes the program to exit before the call to printf even occurs within

the thread. The exit(0) function cancels all execution of all threads. This is a race condition. We do not know whether or not the thread function will be executed first, or if the main program will exit first. On some machines, it’s possible that “Hello World” could indeed be printed, but we do not know for sure. With the call to sleep, however, on almost every machine the program will hit exit(0) before it hits the print statement, so nothing will print.

1. We can use either pthread\_join or pthread\_exit instead of exit(0) to fix this bug.

#include "csapp.h"

void \*thread(void \*vargp);

int main()

{

pthread\_t tid;

Pthread\_create(&tid, NULL, thread, NULL);

**Pthread\_join(tid, NULL); /\* First option to fix the bug \*/**

**pthread\_exit(NULL); /\* Second option to fix the bug \*/**

//exit(0);

}

/\* Thread routine \*/

void \*thread(void \*vargp)

{

Sleep(1);

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

return NULL;

}

**Hello-hw.txt**

In order to pass 3 arguments to the thread function, I created a structure called hellohw that has three variables: the sum that is being calculated, the message that we want printed, and the taskid of the thread. This is due to the fact that the thread function can only have at most one argument, which is of type void. Therefore, we can create a variable of this structure type, cast it to void \* and pass it to the thread function, and then cast that variable back to a variable of the structure type. The structure variable just needs to be populated with the information that is needed for the thread.

In the PrintHello method, I casted the void \* argument to the structure variable type, hellohw, and then populated the three variables using the structure variables in order to print them. In the main function, I added code to initialize each structure, create the threads, and then join them.

#include <pthread.h>

#include <string.h>

#include <stdlib.h>

#include <stdio.h>

#define NUM\_THREADS 8

char \*messages[NUM\_THREADS];

// Struct to hold the 3 arguments that we need for the function

**struct hellohw**

**{**

**int sum;**

**char \* message;**

**int taskid;**

**};**

void \*PrintHello(void \*threadarg)

{

int taskid, sum;

char \*hello\_msg;

**struct hellohw \* hello = (struct hellohw \*) threadarg;**

**sum = hello->sum;**

**hello\_msg = hello->message;**

**taskid = hello->taskid;**

printf("Thread %d %s Sum=%d\n", taskid, hello\_msg, sum);

pthread\_exit(NULL);

}

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

{

pthread\_t threads[NUM\_THREADS];

int rc, t, sum;

sum=0;

messages[0] = "Hello-0";

messages[1] = "Hello-1";

messages[2] = "Hello-2";

messages[3] = "Hello-3";

messages[4] = "Hello-4";

messages[5] = "Hello-5";

messages[6] = "Hello-6";

messages[7] = "Hello-7";

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

{

sum = sum + t;

**// Initialize structure to send to the thread function**

**struct hellohw hello;**

**hello.sum = sum;**

**hello.message = malloc(strlen(messages[t]) \* sizeof(char));**

**strcpy(hello.message, messages[t]);**

**hello.taskid = t;**

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

**// Create the thread. Then wait for the thread to be done.**

**pthread\_create(&threads[t], NULL, PrintHello, &hello);**

**pthread\_join(threads[t], NULL); // Might not be needed**

}

pthread\_exit(NULL);

}

**Dotpr-hw.txt**

In this program, we need to add a mutex in order to allow the threads to safely modify the big\_sum variable. I have created a mutex called mutex, and locked the mutex before modifying the value of big\_sum, and unlocked the mutex after modifying the value of big\_sum. Without the mutex, we could have multiple threads accessing the sum at the same time, causing the incorrect value to be calculated.

In the dotprod function, I added a local variable called offset which will be used for the casted value of the argument. I casted the argument variable to a long. I also added lines to initialize the start and end variables. The start variable was initialized to the value passed to the function times the size of each vector. The end was just the start plus the size of the vector. I initialized the x and y variables to hold array\_a and array\_b respectively. As stated before, I added the mutex lock before modifying the big\_sum variable and mutex unlock after modifying the big\_sum variable.

In the main function, I added code to initialize the mutex. I also added lines to create the thread, and then looped over the threads and joined them with the main program. Finally, I destroyed both the attribute objects used for the p\_thread\_Create and the mutex.

#include <pthread.h>

#include <stdlib.h>

#include <stdio.h>

#define NUMTHRDS 4

#define VECLEN 100000

pthread\_t callThd[NUMTHRDS];

double \*array\_a;

double \*array\_b;

double big\_sum;

int veclen;

**pthread\_mutex\_t mutex; // Mutex for accessing the sum variable**

**pthread\_attr\_t attr;**

void \*dotprod(void \*arg)

{

int i, start, end;

double \*x, \*y;

double mysum;

**long offset = (long) arg;**

// Calculate the starting and ending points for this thread

// Starting point is the current thread number times how many elements

// each thread should do - held in global variable veclen.

**start = offset \* (veclen);**

**// End is just the starting index plus the number of elements**

**end = start + (veclen);**

**x = array\_a;**

**y = array\_b;**

mysum = 0;

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

{

mysum += (x[i] \* y[i]);

}

// Lock the big\_sum variable so that 2 threads don't try to access it at

// the same time.

**pthread\_mutex\_lock(&mutex);** // lock the variable, wait if locked

big\_sum += mysum;

**pthread\_mutex\_unlock(&mutex);** // unlock the variable, allow next thread

return NULL;

}

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

{

long i;

double \*a, \*b;

void \*status;

a = (double\*) malloc (NUMTHRDS\*VECLEN\*sizeof(double));

b = (double\*) malloc (NUMTHRDS\*VECLEN\*sizeof(double));

for (i=0; i<VECLEN\*NUMTHRDS; i++)

{

a[i] = 1;

b[i] = a[i];

}

veclen = VECLEN;

array\_a = a;

array\_b = b;

big\_sum = 0;

**pthread\_mutex\_init(&mutex, NULL); // initialize the mutex**

**pthread\_attr\_init(&attr);**

**pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE);**

/\* create threads \*/

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

{

/\* Each thread works on a different set of data.

The offset is specified by 'i'. The size of

the data for each thread is indicated by VECLEN.

\*/

// Create each thread

**pthread\_create(&callThd[i], &attr, dotprod, (void \*)i);**

}

**/\* Wait on the other threads \*/**

**for (i=0; i < NUMTHRDS; i++)**

**{**

**pthread\_join(callThd[i], NULL); // wait for each thread to be done**

**}**

printf ("Sum = %f \n", big\_sum);

free (a);

free (b);

**pthread\_attr\_destroy(&attr); // destroy the attribute**

**pthread\_mutex\_destroy(&mutex); // destroy the mutex**

pthread\_exit(NULL);

}