Lab report 2

Nhan Dao

October 12, 2020

1. MONTE CARLO'S ESTIMATION OF PI USING POSIX'S PTHREADS

a. Briefly explain the difference between a process and a thread

A process can have multiple threads, these threads are considered within the process by the kernel. Being a part of the same process, threads shares instruction, global, and heap region of memory.

A multi-process program has memory space for each process, hence inter-process communication is slow because memory is not shared between them unlike threads. Context switching between process is more expensive compared to threads - which means when the OS/Kernel switch to executing a different processes, it has to reload the registers to regain the context for that execution, this does not happen with threads.

b. Creating & Joining threads using POSIX

Listing 1 and listing 2 shows the creation and joining of POSIX threads.

```
1  for(i = 0; i < thread_count; i++) {
2    pthread_create(&thread_handles[i], NULL, &Thread_work, (void*) i);
3  }
```

Listing 1: Creating Pthreads (line 45-48 in lab2part1.c)

```
1  for(i = 0; i < thread_count; i++) {
2    pthread_join(thread_handles[i], NULL);
3  }
```

Listing 2: Joining Pthreads (line 49-51 in lab2part1.c)

c. Generating random uniformly distributed dart tosses

Listing 3 shows the subroutine that generates uniformly distributed dart tosses between [-1, 1]. Each worker thread runs this subroutine, and counts the number of tosses that falls within the unit circle.

```
1  srand(seed);
2  for(toss = start; toss < finish; toss++) {
3    x = (double) rand() / RAND_MAX * 2 - 1;
4    y = (double) rand() / RAND_MAX * 2 - 1;
5    distance_squared = x*x + y*y;
6    if (distance_squared <= 1)
7      local_number_in_circle += 1;
8  }
```

Listing 3: Generating uniformly distributed dart tosses (line 82-89 in lab2part1.c)

d. Initializing and destroying Mutex

```
1  pthread_mutex_init(&mutex, NULL);
```

Listing 4: Initializing Mutex (line 42 in lab2part1.c)

```
1  pthread_mutex_destroy(&mutex);
```

Listing 5: Destroying Mutex (line 56 in lab2part1.c)

e. Computing global sum of the number of dart tosses within the unit circle

Listing 6 shows the code that compute the global sum of all 'successful' samples whilst avoiding any race condition using mutex.

```
1  pthread_mutex_lock(&mutex);
2  number_in_circle += local_number_in_circle;
3  pthread_mutex_unlock(&mutex);
```

Listing 6: Computing global sum (line 95-98 in lab2part1.c)

f. Results

Estimating π with Monte Carlo's method using 10^9 tosses with 4 threads vs 1 thread. The linux Time command was used to time the execution.

The estimated values for π was accurate to 3 decimal places. It took 2:13.82s to compute π using 10^9 samples with 4 threads, and only 25.790s to compute with 1 thread per fig. 1.1 and fig. 1.2 respectively.

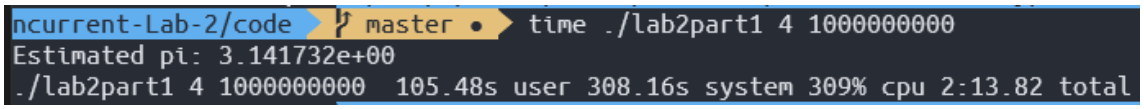

Figure 1.1: Terminal output of lab2part1.c program using 10^9 tosses with 4 threads

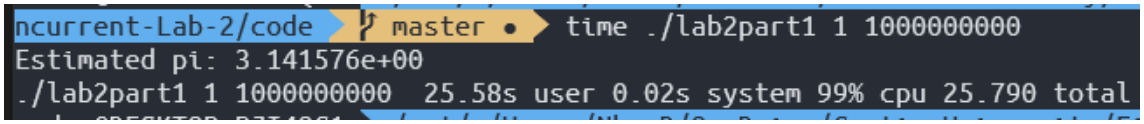

Figure 1.2: Terminal output of lab2part1.c program using 10^9 tosses with 1 threads

2. Trapezoid rule with POSIX Pthreads

a. Creating threads using POSIX

```
1     /* Start the threads. */
2     for (i = 0; i < thread_count; i++) {
3         pthread_create(&thread_handles[i], NULL, &Thread_work, (void*)
              i);
4     }
```

Listing 7: Creating POSIX Pthreads (line 72-75 in lab2part2.c)

b. Joining threads using POSIX

```
1     /* Wait for threads to complete. */
2     for (i = 0; i < thread_count; i++) {
3         pthread_join(thread_handles[i], NULL);
4     }
```

Listing 8: Joining POSIX Pthreads (line 77-80 in lab2part2.c)

c. If each thread is allowed to increment the total sum on the fly, does a race condition exist? If so, how does this affect the final result?

With syncronisation, a race condition will exist when multiple thread increment the total sum at the same time. This affects teh final result by making the global total sum to be less than the actual sum due increments that were unaccounted for because they were done at the same time.

d. Mutex based critical section

```
1          pthread_mutex_lock(&mutex);
2          total += my_int;
3          pthread_mutex_unlock(&mutex);
```

Listing 9: Mutex based threads synchronisation (line 120-122 in lab2part2.c)

e. Semaphore based critical section

```
1          sem_wait(&sem);
2          total += my_int;
3          sem_post(&sem);
```

Listing 10: Semaphores based threads synchronisation (line 120-122 in lab2part2.c)

f. Busy-wait based critical section

```
1          while (flag != my_rank);
2          total += my_int;
3          flag = (flag + 1) % thread_count;
```

Listing 11: Busy-wait based threads synchronisation (line 115-117 in lab2part2.c)

g. Results

The code was compile to compute the trapezoid rule for $a = 0$, $b = 1$, $n = 2^{30}$ for the function $f(x)$. 128 threads were used for each of the three methods (mutex, semaphores, busy-wait) to determine whic method is the fastest. The function $f(x)$ is hardwired to compute x^2, see listing 12.

```
1 /*--------------------------------------------------------------------*/
2 double f(double x) {
3     double return_val;
4
5     return_val = x*x;
6     return return_val;
7 } /* f */
```

Listing 12: Hardwired f(x) for trapezoid rule (line 154-159 in lab2part2.c)

Linux echo was use to pipe in the user input so that it does not vary execution the execution time. Linux's time command was used to time the different synchronisation methods. The results are shown in figs. 2.1 to 2.3.

The result shows that semaphores is the fastest, then mutex and then busy-wait in terms of execution time. Busy wait is the least efficient because it consumes CPU cycles as the threads continously wait. Busy-wait enfores the order of access to the critical section, and waits for the

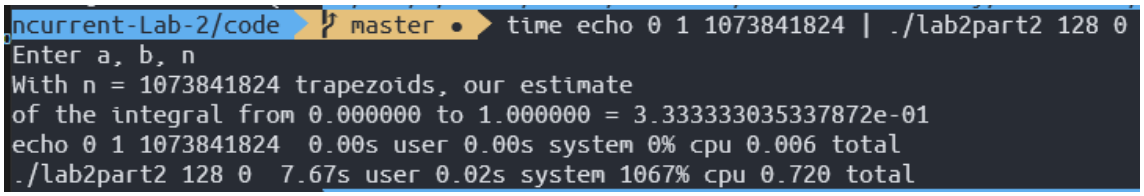

Figure 2.1: Terminal output of lab2part2.c program using 128 threads and mutex

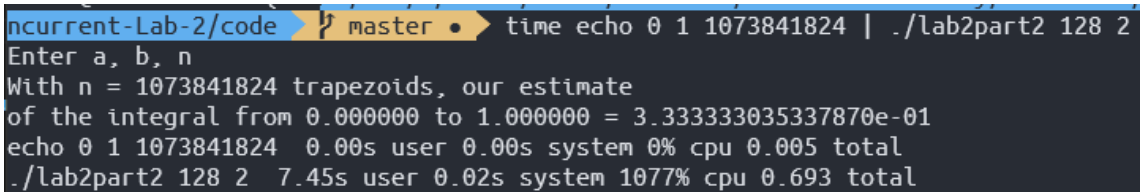

Figure 2.2: Terminal output of lab2part2.c program using 128 threads and semaphores

system to schedule on the thread rank matching the flag variable to allow be allowed through - if it schedules on the wrong thread (a thread that is busy-waiting), it wastes CPU cycles in the checking of the while condition until it deschedules and eventually until it picks the correct thread.

Mutex is middle between the three, only slower than semaphores by 40ms. The order the threads execute the critical section is random, which does not guarentee that locks are given in the order they are called.

3. Multi-threaded file tokenizer

a. Run with 2 threads and the input file provided. Does the code run exhibiting the desired behaviour?

The code does not exhibit the correct behaviour because the threads are not tokenizing the lines in a round-robin fashion. This cause unbalanced loading of work for each threads and means that some thread may be idle waiting for work. This is exactly the case as seen in fig. 3.1, only a single line is allocated to thread1, and the rest of the lines in the file is allocated to thread0 to tokenize.

b. Inital value of samaphores for each thread

The number of semaphores are the same as the number of threads created using an n array (sem_t) array where n is the number of threads. The first semaphore in position 0 is initialized to 1 and the rest are intialized to 0, see listing 13.

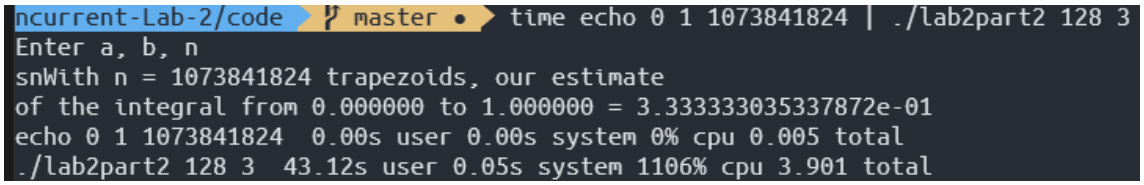

Figure 2.3: Terminal output of lab2part2.c program using 128 threads and busy-wait

```
1    sems = (sem_t*) malloc(thread_count*sizeof(sem_t));
2    sem_init(&sems[0], 0, 1);
3    for (thread = 1; thread < thread_count; thread++)
4       sem_init(&sems[thread], 0, 0);
```

Listing 13: Initial values for each semaphores for each thread (line 37-40 in lab2part3.c)

c. Modifying *Tokenize* function to achieve the desired behaviour

To allow for the threads to read each line of the file in a round-robin fashion, semaphores can be used to synchronise the thread. Essentially, a thread with rank i call *sem_wait* on semaphore i before entering the critical section, which is reading the new line from the file pointer and executing the tokenizer. Then it calls *sem_post* on sem $i+1$, which means that the thread with rank $i+1$ can grab the nextline, (modulo the result by the total thread count to wrap the largest rank thread back to the thread with rank 0). Listing 14, shows the section of code in *Tokenize* that performs the load balancing.

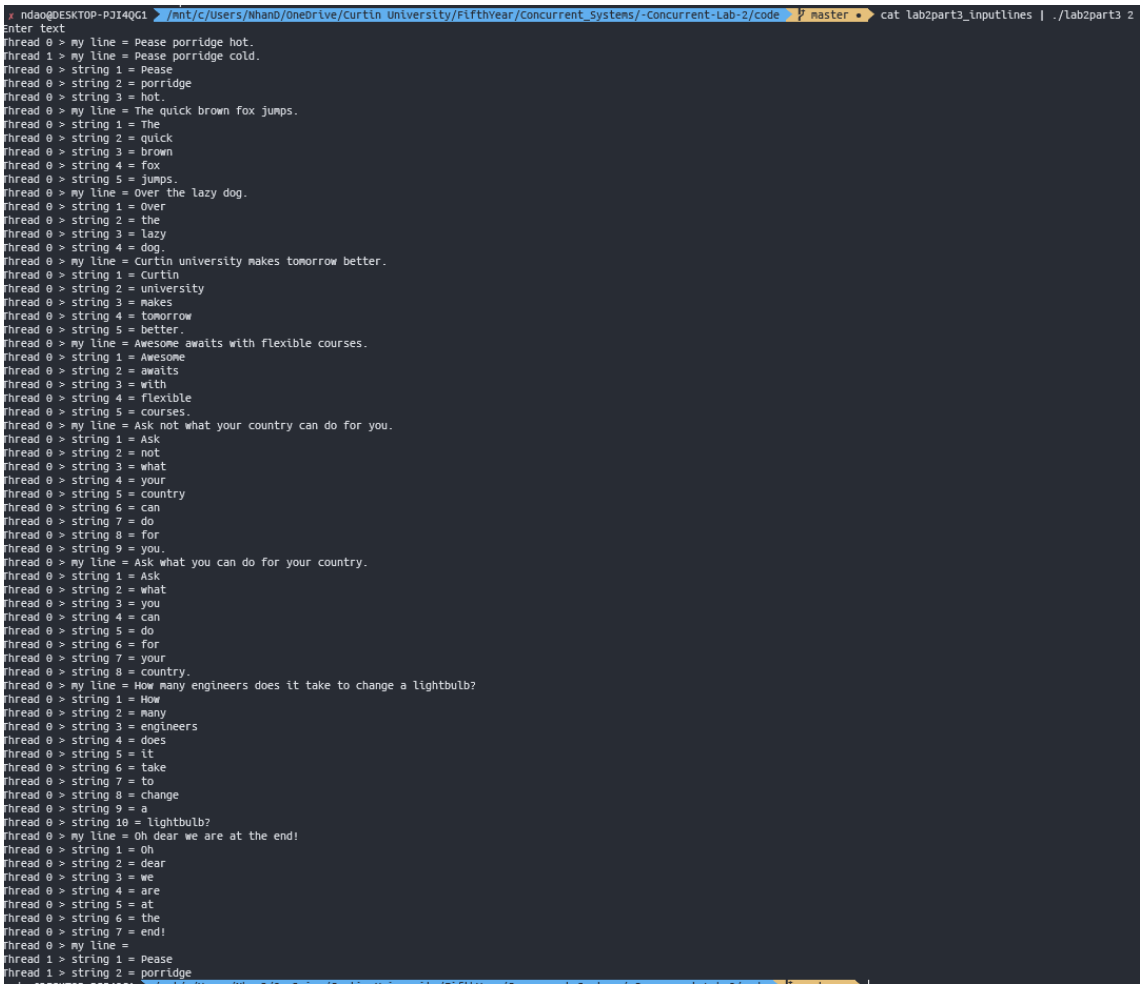

Figure 3.1: Terminal output of lab2part3.c program using 3 threads and lab2part3_inputlines as input

```
1   /* Have each thread read consecutive lines and tokenize them in turn
        */
2   sem_wait(&sems[my_rank]);
3   fg_rv = fgets(my_line, MAX, stdin);
4   sem_post(&sems[(my_rank + 1) % thread_count]);
5   while (fg_rv != NULL) {
6       printf("Thread %ld > my line = %s", my_rank, my_line);
7
8       count = 0;
9       next_string = my_line;
10      my_string = my_strtok(" \t\n", &next_string);
11      while ( my_string != NULL ) {
12          count++;
13          printf("Thread %ld > string %d = %s\n", my_rank, count,
                my_string);
14          free (my_string);
15          my_string = my_strtok(" \t\n", &next_string);
16      }
```

```
17          sem_wait(&sems[my_rank]);
18          fg_rv = fgets(my_line, MAX, stdin);
19          sem_post(&sems[(my_rank + 1) % thread_count]);
20      }
```

Listing 14: Thread syncrhonsation using semaphores for load balancing (line 131-151 in lab2part3.c)

4. PRODUCER-CONSUMER

a. What information is being written into and read from the buffer (in lab2part4.c)?

The producer is writting into the buffer the count to the number of iterations *i*, and the consumer is attempts to read the value and sets the buffer to zero.

b. Describe in detail the action of the producer and consumer thread as well as the flow of execution.

The *producer* enters the critical section and attempts to sets the buffer to the value *i*, which is the current loop iteration - if the value of the buffer is non-zero, it means that the consumer has not read from the buffer. This trigger a conditional wait, where the consumers releases the lock, and awaits for a signal. When the buffer is ready to be modify, the *producer* sets the buffer to the value of *i* and then signals *condc* which tells the *consumer* that there's item in the buffer.

Vice versa, as the *consumer* enters its critical section and attempts to read the buffer - if the buffer is zero means that it's empty, then the consumer enters a conditional wait for the *condc* signal. When the buffer is non-zero, it then sets the value of the buffer to zero, and signal the *condp* signal to notify the *producer* thread to place item in the buffer.

c. Is there a race condition? Fully justify your answer

The race-condition is mitigated by the mutex lock. The producer-consumer have shared memory in the form of a buffer. Without the mutex lock, both threads can modify the buffer at the same time which is a race condition.

d. What is the producer thread doing while the consumer thread is active and vice-versa?

.

The producer thread is waiting for the *condp* signal from the consumer, which tells the producer thread to exit the conditional wait loop and modify the buffer.

Vice-versa, the consumer thread is waiting for the *condc* signal from the producer, which tells the consumer that theres item in the buffer.

e. What is the value in *buffer* just before the main function terminates and will it always be the same?

It will always be zero because the consumer and producer thread knows before hand how many times buffer is modified in the form of the MAX variable.