Discussion 11

Aniket Shivam

Quick reminder

- Lab Assignment 1 is due on Monday!

- Office Hours:
  - Both Lab Sections are treated as Office Hours.
  - Via Zoom
  - Plus post questions on Piazza

What is a Data Race?

Occurs during the execution of a multi-threaded process/application.

A data race occurs when:

- Two or more threads in a **single process** access the same memory location concurrently, and
- At least one of the accesses is for writing, and
- The threads are not using any exclusive locks (like mutex) to control their accesses to that memory.

When these three conditions hold, the order of accesses is non-deterministic, and the computation may give different results from run to run depending on that order.

Some data-races may be benign (for example, when the memory access is used for a busy-wait), but many data-races are bugs in the program.

Simple Example - Data Races

- What's wrong with the code:
    - 'Global' variable is being updated by multiple threads.
    - Satisfy all three conditions for the data race.
- To read more about data races examples:
    - https://github.com/google/sanitizers/wiki/ThreadSanitizerPopularDataRaces

```c
#include <pthread.h>
#include <stdio.h>

int Global;

void *Thread1(void *x) {
  Global++;
  return NULL;
}

void *Thread2(void *x) {
  Global--;
  return NULL;
}

int main() {
  pthread_t t[2];
  pthread_create(&t[0], NULL, Thread1, NULL);
  pthread_create(&t[1], NULL, Thread2, NULL);
  pthread_join(t[0], NULL);
  pthread_join(t[1], NULL);
}
```

How can you detect it?

- Load gcc/8.2.0
- While compiling your program, enable Google's Thread Sanitizer (Runtime Library).
  - https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual
  - Add flags: -fsanitize=thread -fPIE -pie
- If data races are detected, then warnings will be produced while running your application.
- Otherwise, no execution time warnings.

Simple Example - Data Races

```
$ g++ tsan_test.cpp -o tsan_test -pthread -fsanitize=thread -fPIE -pie -g

$ ./tsan_test
==================
WARNING: ThreadSanitizer: data race (pid=10940)
  Read of size 4 at 0x56081df73094 by thread T2:
    #0 Thread2(void*) /home/aniketsh/tsan_test.cpp:12 (a.out+0xb9d)

  Previous write of size 4 at 0x56081df73094 by thread T1:
    #0 Thread1(void*) /home/aniketsh/tsan_test.cpp:7 (a.out+0xb5e)

  Location is global 'Global' of size 4 at 0x56081df73094 (a.out+0x000000202094)

  Thread T2 (tid=10943, running) created by main thread at:
    #0 pthread_create ../../../../gcc-8.2.0/libsanitizer/tsan/tsan_interceptors.cc:915 (libtsan.so.0+0x2af7b)
    #1 main /home/aniketsh/tsan_test.cpp:19 (a.out+0xc1f)

  Thread T1 (tid=10942, finished) created by main thread at:
    #0 pthread_create ../../../../gcc-8.2.0/libsanitizer/tsan/tsan_interceptors.cc:915 (libtsan.so.0+0x2af7b)
    #1 main /home/aniketsh/tsan_test.cpp:18 (a.out+0xbfe)

SUMMARY: ThreadSanitizer: data race /home/aniketsh/tsan_test.cpp:12 in Thread2(void*)
==================
ThreadSanitizer: reported 1 warnings
```

Simple Example - Data Races

- A solution is using mutex.
- Result: No runtime warning.

```
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t m;

int Global;

void *Thread1(void *x) {
  while (pthread_mutex_trylock(&m) !=0);
  Global++;
  pthread_mutex_unlock(&m);
  return NULL;

}

void *Thread2(void *x) {
  while (pthread_mutex_trylock(&m) !=0);
  Global--;
  pthread_mutex_unlock(&m);
  return NULL;
}

int main() {
  pthread_t t[2];
  pthread_create(&t[0], NULL, Thread1, NULL);
  pthread_create(&t[1], NULL, Thread2, NULL);
  pthread_join(t[0], NULL);
  pthread_join(t[1], NULL);
}
```

Dining Philosophers Output

Use philosopher data struct for recording times inside Pthread functions

time(&phil->meal_time[phil->course]);

After joining all the threads, print meal times for all philosophers.

philosophers[i].meal_time[j] - start_time

Sample Output:

```
Creating thread 0
Creating thread 1
Creating thread 2
Joined all threads
Philosopher 0 ate meal 0 at 0
Philosopher 0 ate meal 1 at 1
Philosopher 0 ate meal 2 at 4
Philosopher 1 ate meal 0 at 3
Philosopher 1 ate meal 1 at 7
Philosopher 1 ate meal 2 at 8
Philosopher 2 ate meal 0 at 2
Philosopher 2 ate meal 1 at 5
Philosopher 2 ate meal 2 at 6
```

Dynamic Allocation of Chunks for Part B Edge Detection

Guidelines to Parallelize the Code:

- Two of the command line parameters are: Number of Threads and Number of Chunks.
- Number of Threads signify the total threads to be created by the program.
- Number of Chunks signify the number of equal size partitions of the input images.
    - For example: if Input image has 100 rows (i.e. height), and Number of Chunks is 4, then each chunk will have 25 rows each.
    - If for the same image, Number of Chunks is 6, then there will 5 partitions of 17 rows each (rounding up 100 divided by 6) and one last partition for the remaining rows, i.e., 15 rows.

Dynamic Allocation of Chunks for Part B Edge Detection

- **Scheduling of chunks on the threads:**
  - We will do dynamic allocation of chunks to the threads.
  - Since this is dynamically decided (at execution), each thread should track if there are any chunk that hasn't been processed and fetch one. (HINT: Mutex and a global variable)
  - Result of dynamic scheduling is that on each execution different threads will get different chunks.
    - This happens because creation of threads and processing of chunks can take different time for different threads.
  - For example: if there are 6 chunks and 4 threads.
    - First Execution: Thread 1 processes chunk 1, 5; Thread 2 processes chunk 2; Thread 3 processes chunk 3, 6; Thread 4 processes chunk 4.
    - Second Execution: Thread 1 processes chunk 2; Thread 2 processes chunk 3, 6; Thread 3 processes chunk 1, 5; Thread 4 processes chunk 4.

Questions?

You can post on Piazza
Or
Attend Office hours

Discussion Slides will be posted on Class Website