CSL301
12340220

Assignment: TLB and Page Fault Measurement in XV6

Question 1.

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

int global = 10;

void* threadFunction(void* arg) {
    int id = *(int*)arg;
    int local_thread = id * 100;

    printf("Thread %d:\n", id);
    printf("    Address of global: %p\n", (void*)&global);
    printf("    Address of local_thread : %p\n\n", (void*)&local_thread);

    return NULL;
}

int main() {
    pthread_t t1, t2;
    int id1 = 1, id2 = 2;
    int local_main = 20;

    printf("Main thread:\n");
    printf("    Address of global: %p\n", (void*)&global);
    printf("    Address of local_main: %p\n\n", (void*)&local_main);

    pthread_create(&t1, NULL, threadFunction, &id1);
    pthread_create(&t2, NULL, threadFunction, &id2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

Global variables are stored in code part of the address space which is the same for every thread, thus they have the same address,
Local variables are stored in stack part of the address space, which is different for each thread, thus they have different addresses

Output:
Main thread:
Address of global: 0x5d985881f010
Address of local_main: 0x7ffc6b7b4364

Thread 1:
Address of global: 0x5d985881f010
Address of local_thread : 0x799532dfbeb0

Thread 2:
Address of global: 0x5d985881f010
Address of local_thread : 0x7995325faeb0

```
Main thread:
    Address of global: 0x5d985881f010
    Address of local_main: 0x7ffc6b7b4364

Thread 1:
    Address of global: 0x5d985881f010
    Address of local_thread : 0x799532dfbeb0

Thread 2:
    Address of global: 0x5d985881f010
    Address of local_thread : 0x7995325faeb0
```

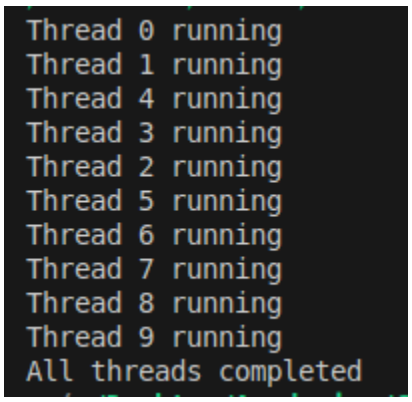
Question 2.

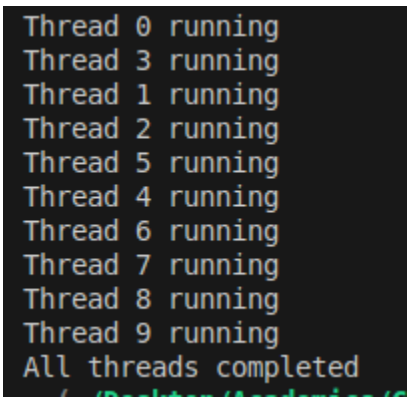
```c
#include <stdio.h>
  #include <pthread.h>
  #include <stdlib.h>


  void* thread_function(void* arg) {
      int thread_num = *(int*)arg;
      printf("Thread %d running\n", thread_num);
      return NULL;
  }

  int main() {
      int NUM_THREADS = 10;
```

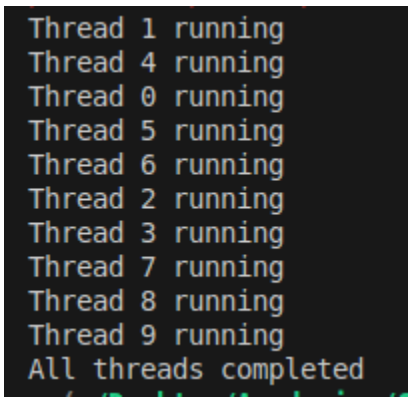
```c
    pthread_t threads[NUM_THREADS];
    int thread_nums[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        thread_nums[i] = i;
        if (pthread_create(&threads[i], NULL, thread_function, &thread_nums[i]) != 0) {
            perror("Failed to create thread");
            exit(1);
        }
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("All threads completed\n");
    return 0;
}
```

Explanation:
The order may vary between runs, because thread scheduling is controlled by the **OS scheduler**.
Each thread might start or get CPU time at different moments.
Even though threads are created in a loop, the OS decides when each one runs.

Output:







Different order in every run

Question 3.

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

long long counter = 0;

void* increment_counter(void* arg) {
    for (int i = 0; i < 10000000; i++) {
        counter++;
    }
    return NULL;
}

int main() {
    int NUM_THREADS = 10;
    pthread_t threads[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, increment_counter, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Final counter value: %lld\n", counter);
    printf("Expected value: %lld\n", (long long)NUM_THREADS * 10000000);

    return 0;
}
```

Explanation:
The final value differs because multiple threads modify the shared variable counter at the same time without synchronization.
The expression: counter++ is not atomic, it actually involves three steps:

1. Read the value of counter
2. Add 1
3. Write the new value back to memory

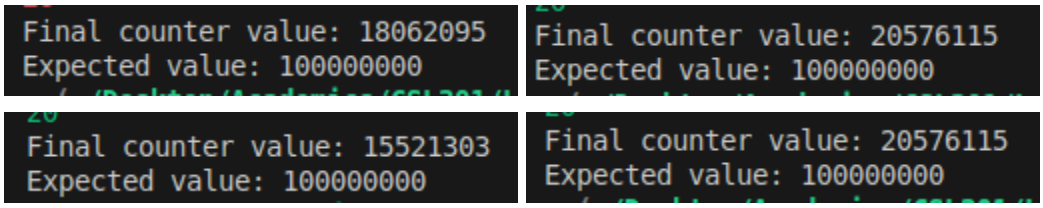
When two threads do this simultaneously, their operations can overlap. That's why the final result is smaller than expected.

A race condition occurs when

- Two or more threads access shared data at the same time,
- At least one thread modifies that data,
- And the final result depends on the unpredictable timing of their execution.

It leads to incorrect and non-deterministic output

Output:



Question 4.

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

void* compute_square(void* arg) {
    int n = *(int*)arg;
    int* result = malloc(sizeof(int));
    *result = n * n;
    return result;
}

int main() {
    pthread_t tid;
    int n;

    printf("Enter a number: ");
    scanf("%d", &n);
    if (pthread_create(&tid, NULL, compute_square, &n) != 0) {
        perror("Failed to create thread");
        return 1;
    }
    void* result;
    pthread_join(tid, &result);
    printf("Square of %d is %d\n", n, *(int*)result);
    free(result);

    return 0;
}
```

Explanation:
pthread_create() starts a new thread and passes the address of num as an argument.

The thread function compute_square() reads the integer, computes its square, allocates memory using malloc() for the result (so it survives after the thread ends), and then returns the pointer to that memory

The main thread calls: pthread_join(tid, &result);
This waits for the thread to finish and stores the return pointer in result.

Output:

```
Enter a number: 400
Square of 400 is 160000
```

Question 5.

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

void* compute_square(void* arg) {
    int n = *(int*)arg;
    int* result = malloc(sizeof(int));
    *result = n * n;
    return result;
}

int main() {
    int n;
    printf("Enter number of threads: ");
    scanf("%d", &n);

    pthread_t threads[n];
    int thread_nums[n];
    void* result;
    int sum = 0;

    for (int i = 0; i < n; i++) {
        thread_nums[i] = i + 1;
        if (pthread_create(&threads[i], NULL, compute_square, &thread_nums[i]) != 0) {
            perror("Failed to create thread");
            return 1;
        }
    }

    for (int i = 0; i < n; i++) {
        pthread_join(threads[i], &result);
```

```
            int value = *(int*)result;
            printf("Thread %d returned %d\n", i + 1, value);
            sum += value;
            free(result);
        }

        printf("Sum of all threads: %d\n", sum);
        return 0;
    }
```

Explanation:

Each thread receives its thread number (1 to n).

It computes the square of that number and returns it via pthread_exit() (implicit via return).
The main thread:

1. Calls pthread_join() for each thread.
2. Collects the returned value.
3. Prints per-thread results and accumulates the total sum.

Output:

```
Enter number of threads: 10
Thread 1 returned 1
Thread 2 returned 4
Thread 3 returned 9
Thread 4 returned 16
Thread 5 returned 25
Thread 6 returned 36
Thread 7 returned 49
Thread 8 returned 64
Thread 9 returned 81
Thread 10 returned 100
Sum of all threads: 385
```