Threads and Locks

Welcome to Threads

Up until now, most of your programs have involved only a single thread. Within this lab we will explore initialising threads and writing separate thread functions. Given the following segment, you should be able to observe the output from the program below. Update the code to output thread id more than two times.

```
struct thread_data {
    int tid;
};

void* work1(void* arg) {
    struct thread_data* data = (struct thread_data*) arg;
    printf("Hello from thread %d\n", data->tid);
    return NULL;
}

int main() {
    struct thread_data data = { { 1 }, { 2 } };
    pthread_t threads[2];
    pthread_create(threads, NULL, work1, &data);
    pthread_create(threads+1, NULL, work2, &data);
    printf("main thread has created threads\n");
    pthread_join(threads[0], NULL);
    pthread_join(threads[1], NULL);
    printf("program finished\n");
    return 0;
}
```

Discuss with your class the following. * How is data passed to the thread? * How is the thread started * How do we rejoin the thread with the main thread? * What is mutable and immutable data?

Parallel Sum

Write a program that will compute the sum of an array. Divide the workload between threads and correctly produce the result.

- Does your program share memory? If so, what kind of memory is being shared (immutable or mutable?)
- What are the risks of sharing mutable memory?
- Does your implementation produce the correct result? Explain why/why not

False Sharing

Given the following segment of code, examine the performance, do you notice any issues when we increase the number of threads. Does the increase of threads correspond with an increase of performance or not?

```c
struct thread_data {
    int* result;
    int result_index;
    int* numbers;
    int numbers_len;
};

void* worker(void* data) {
    struct thread_data* d = (struct thread_data*) d;
    for(int i = 0; i < d->numbers_len; i++) {
        d->result[d->result_index] += d->numbers[i];
    }
    return NULL;
}
```

- What issues can arise when you share data among threads?
- If you were use a global variable instead of separate thread data objects, what issues could arise?
- When data is shared between threads, do you notice a speed up with multiple threads?

Ruining your linked list

Design a program that will share a linked list among multiple threads, where each thread will add elements to the linked list. Run the program and observe the results, if it happens to finished, your program should attempt to traverse the linked list and ensure that all elements were added.

- What did you observe with your program?
- If a problem is occuring, what could causing it?

Coarse-grained locking

Use your linked list that you have constructed previously or you could construct a dynamic array for this task if you had not completed the linked list. The following functions must be thread safe.

```c
void linkedlist_map_put(struct linkedlist_map* map,
    void* key void* value);
void* linkedlist_map_get(struct linkedlist_map* map,
    void* key);
```

```
void* linkedlist_map_remove(struct linkedlist_map* map,
    void* key);
```

Hoarder and Making

Unfortunately we can't have multiple threads working on our linked list, however we can ensure that any data sent to our linked list is by one thread. For this problem you are required to give roles to two threads. * Thread 1 will be responsible for accepting storing data sent to it, it will use a linked list map to store the data * Thread 2 will be responsible for sending data to Thread 1

As implied by the title of the question, one thread is a hoarder and the other is a maker. Use a pipe to satisfy this constraint.