# Synchronisation 1

## Shared Memory and Locking

Below is an example of two threads accessing shared data. Compile this program and observe the final value.

```c
struct thread_data {
    int value;
};

void* work(void* arg) {
    struct thread_data* data = (struct thread_data*) arg;
    for(int i = 0; i < 1000000; i++) {
        data->value += 1;
    }
    return NULL;
}

int main() {
    struct thread_data data = { 0 };
    pthread_t threads[2];
    pthread_create(threads, NULL, work, &data);
    pthread_create(threads+1, NULL, work, &data);

    for(int i = 0; i < 2; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("%d\n", data.value);
    return 0;
}
```

- When running this program multiple times, is the final result consistent?
- If you have observed the final value does not meet the expected result, discuss why this could be happening
- How could ensure that the final result is what we expect? What modifications do we need to make?

## Dining Philosophers

Assume that there are N philosophers sitting at a round table. A single chopstick is placed between two adjacent philosophers. Every philosopher is either thinking or eating. However, a philosopher needs both chopsticks (to the left and to the right) to start eating. They are not allowed to acquire chopsticks that are not immediately adjacent to them. Complete the following program so that each philosopher is able to eat.

```c
struct args {
    pthread_mutex_t lock;
    unsigned int id;
};

void* dine(void* arg) {
    const unsigned id = *((unsigned *) arg);
    while (true) {
        // TODO: Acquire two chopsticks first
        // the ith philosopher can only reach
        // the ith and (i + 1)th chopstick
        printf("Philosopher %u is eating\n", id);
    }
    return NULL;
}

int main(void) {
    unsigned args[THINKERS];
    pthread_t thinkers[THINKERS];
    pthread_mutex_t chopsticks[THINKERS];

    // create the chopsticks
    for (size_t i = 0; i < THINKERS; i++) {
        if (pthread_mutex_init(chopsticks + i, NULL) != 0) {
            perror("unable to initialize mutex");
            return 1;
        }
    }

    // launch threads
    for (size_t i = 0; i < THINKERS; i++) {
        args[i] = i;
        if (pthread_create(thinkers + i, NULL, dine, args + i) != 0) {
            perror("unable to create thread");
            return 1;
        }
    }

    // wait for threads to finish
    for (size_t i = 0; i < THINKERS; i++) {
        if (pthread_join(thinkers[i], NULL) != 0) {
            perror("unable to join thread");
            return 1;
        }
    }
    // remove the chopsticks
```

```
    for (size_t i = 0; i < THINKERS; i++) {
        if (pthread_mutex_destroy(chopsticks + i) != 0) {
            perror("unable to destroy mutex");
            return 1;
        }
    }
    return 0;
}
```

## Semaphore Philosophers

We have previously solved the dining philosophers problem by using a locking hierarchy. This time, use a semaphore for the table that only allows N/2 philosophers to eat at a time.

## Fine-Grained Locking

In a previous tutorial, you were asked to construct a simple mutex around a linked list, however, we want to apply more fine-grained locking on our linked list.

Add a mutex field to each node and ensure that on any access of a linked list node, the predecessor is also locked and updated accordingly.

- For any get/read operation, any value returned is assumed to have been borrowed. To properly gaurantee the lifetime of a value associated with the linked list, you will need to construct a reference counting mechanism.

Refer to next week's optional exercise with atomics.

- For any removal or put operation, the nodes must be locked for the modification to occur.
- destroy function are assumed to not be thread-safe and requires the user to ensure that other threads are no longer modifying the data structure.

## Atomic Operations

As we have noted prior, race conditions occur when we have more than one thread mutating shared memory.

Atomics actually allow us to deal with this situation by making the write visible to other threads, this occurs in a single operation on the hardware itself.

Simply start by writing a program where multiple threads will increment the shared counter.

```
#include <stdatomic.h>
atomic_int counter = 0;
```

```
void increment() {
    atomic_fetch_add(&counter, 1);
}
```

- Set up your threads and a worker function to continually increment the counter, observe the result at the end of execution
- What do you observe with this code?
- Do you observe significant performance differences between single threaded and multi-threaded performance?

## Test and Set Lock

Using what you have learned from atomic operations and write your own lock mechanism. Use the following scaffold to start building your test and set spinlock.

```
struct tas;

/**
* @pram taslock, initialises the handle for the taslock
*/

void tas_init(struct tas*);
/**
* Locks, operation should be successful if the lock is valid.
* If the lock is invalid, the lock operation returns a non-zero integer
* tas lock that is currently in a locked state will keep threads waiting.
* @param taslock
* @return success
*/
int tas_lock(struct tas* t);
/**
* Unlocks, operation should be successful is the lock is valid.
* If the lock is invalid, unlock operation returns a non-zero integer.
* @param taslock
* @return success
*/
int tas_unlock(struct tas* t);
/**
* Destroys the tas lock, puts it in an invalid state
* @param taslock
*/
void tas_destroy(struct tas* t);
```

If the test and set lock has been initialised, the lock and unlock functions shold return the value 0 to indicate a success. Non-zero is reserved for errors regarding the lock.

- Construct your test and set spinloc
- Construct a test suite to ensure that it can handle multiple threads and applied to normal lock object