Prelab

1. What is the underrlying type of a pthread_t? What is its meaning? When is a pthread_t unique? When may it be reused?

   The underlying type for pthread_t is unsigned long int.

   Pthread_t is an abbreviation for “POSIX thread”, which is a thread of execution with a programing that conforms to the POSIX standard for threads. A pthread is an opaque data type that represents a thread in C programming.

   Pthread_t is not inherently unique. The pthread_t data type is typically implemented as an integer or a pointer to a structure that represents the thread, but the exact representation is implementation-dependent and can vary across different systems. However, when you create a new pthread using pthread_create, the resulting pthread_t identifier is guaranteed to be unique within the current process. This means that no two threads within the same process will have the same pthread_value.

   A pthread can be reused after it has terminated, and its resources have been cleaned up using the pthread_join function. This means that the pthread identifier associated with the terminated thread can be used again to create a new thread using pthread_create.

2. What is a thread-safe function? Provide an example implementation of a simple C function that is not thread-safe and an improved version that is thread-safe.

   A thread-safe function is a function that can be safely called from multiple threads simultaneously without causing data races or other synchronization problems. In other words, a thread-safe function is designed to work correctly regardless of how many threads are calling it and in what order.

   Example of not thread-safe C function

```c
#include <stdio.h>

void increment_counter(int *counter) {
    (*counter)++;
}

int main() {
    int counter = 0;

    // create two threads that increment the counter
    // multiple times
    // (not shown for brevity)

    // print the final value of the counter
    printf("Final value: %d\n", counter);
```

```
    return 0;
}
```

Example of thread safe C function

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

pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;

void increment_counter(int *counter) {
    pthread_mutex_lock(&counter_mutex);
    (*counter)++;
    pthread_mutex_unlock(&counter_mutex);
}

int main() {
    int counter = 0;

    // create two threads that increment the counter
    // multiple times
    // (not shown for brevity)

    // print the final value of the counter
    printf("Final value: %d\n", counter);

    return 0;
}
```

3. Given the following C structure:

```
struct foo
{
    unsigned int     f_count;
    pthread_mutex_t  f_lock;
};
```

a. Implement a function void increment(struct foo *), which increments the f_count field in a thread-safe manner.

```
#include <pthread.h>

struct foo {
    unsigned int f_count;
    pthread_mutex_t f_lock;
};
```

```
void increment(struct foo *f) {
    pthread_mutex_lock(&f->f_lock);   // acquire the lock
    f->f_count++;                     // increment the count
    pthread_mutex_unlock(&f->f_lock); // release the lock
}
```

b. Implement the function struct foo * foo_create(void).

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

struct foo {
    unsigned int f_count;
    pthread_mutex_t f_lock;
};

struct foo *foo_create(void) {
    struct foo *f = malloc(sizeof(struct foo));
    if (f == NULL) {
        return NULL; // allocation failed
    }
    f->f_count = 0;
    pthread_mutex_init(&f->f_lock, NULL); // initialize the lock
    return f;
}
```

4. Read the first three sections of Apple's libdispatch tutorial (up to, and including, "Defining work items: Functions and Blocks"). How do the C blocks described in this document compare to C++ lambda functions?

The C blocks described in Apple's libdispatch tutorial and C++ lambda functions are similar in that they both provide a way to define inline functions or closures that can be passed as arguments to other functions. Both constructs capture local variables by value or by reference, and can be used to simplify code and improve readability. However, there are some differences between C blocks and C++ lambda functions. One key difference is the syntax used to define the closures. In C, a block is defined using the **^{}** syntax . In contrast, a C++ lambda function is defined using the **[](){}** syntax. Another difference is the way that variables are captured. In C blocks, variables are captured using the **__block** storage qualifier, which allows them to be modified within the block. In C++ lambda functions, variables are captured using the **[&]** or **[=]** syntax to capture variables by reference or by value, respectively.