LAB 11
Breeha Qasim

3.1a

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

void* function1(void* arg)
{
    printf("Hello World!\n");
}

void* function2(void* arg)
{
    printf("Hello Function2!\n");
}

void* function3(void* arg)
{
    printf("Hello Function3!\n");
}

void* function4(void* arg)
{
    printf("Hello Function4!\n");
}

int main()
{
    //creating multiple threads
    pthread_t thread1, thread2, thread3, thread4;

    // create new threads and assigning different functions to each
    pthread_create(&thread1, NULL, function1, NULL);
    pthread_create(&thread2, NULL, function2, NULL);
    pthread_create(&thread3, NULL, function3, NULL);
    pthread_create(&thread4, NULL, function4, NULL);

    // wait until the thread completes
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    pthread_join(thread4, NULL);

    printf("All threads have completed their tasks.\n");

    return 0;
}
```

Explanation:
This code shows how to create and execute multiple threads. Each of the four functions defined prints a different message. Four threads are generated in the main function, and each thread is tasked with carrying out a given function. In order to make sure that every thread completes its task before the main program ends, the program then uses pthread_join() function to wait for all threads to finish. This final message is printed after every thread has finished. Basic thread creation, execution, and timing are demonstrated in this code.

3.1b

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

void* function1(void* arg)
{
    printf("Hello World from Function1! Thread ID (TID): %lu, Process ID (PID): %d\n", pthread_self(), getpid());
    return NULL;
}

void* function2(void* arg)
{
    printf("Hello from Function2! Thread ID (TID): %lu, Process ID (PID): %d\n", pthread_self(), getpid());
    return NULL;
}

void* function3(void* arg)
{
    printf("Hello from Function3! Thread ID (TID): %lu, Process ID (PID): %d\n", pthread_self(), getpid());
    return NULL;
}

void* function4(void* arg)
{
    printf("Hello from Function4! Thread ID (TID): %lu, Process ID (PID): %d\n", pthread_self(), getpid());
    return NULL;
}

int main()
{
    //creating multiple threads
    pthread_t thread1, thread2, thread3, thread4;
    // create new threads and assign different functions to each
    pthread_create(&thread1, NULL, function1, NULL);
    pthread_create(&thread2, NULL, function2, NULL);
    pthread_create(&thread3, NULL, function3, NULL);
    pthread_create(&thread4, NULL, function4, NULL);
    // wait until each thread completes
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    pthread_join(thread4, NULL);

    printf("All threads have completed their tasks.\n");
    return 0;
}
```

Explanation:
Four threads are created by this program, each of which runs a distinct function function1 to function4 and outputs its thread and process IDs. The pthread_join function calls make sure each thread finishes before the program terminates with a final message verifying all tasks have been completed, following the pthread_create calls that started all threads.

4.1

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

// Function to print a string, TID, and PID
void *print_message(void *args)
{
    char *str = (char *)args;
    printf("Message: %s\n", str);
    printf("TID: %lu, PID: %d\n", pthread_self(), getpid());
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t t1, t2, t3, t4;  // thread identifiers

    // strings to pass to the threads
    char msg1[] = "Hello World!";
    char msg2[] = "Hello from thread 2!";
    char msg3[] = "Hello from thread 3!";
    char msg4[] = "Hello from thread 4!";

    // create threads with different string arguments
    pthread_create(&t1, NULL, print_message, msg1);
    pthread_create(&t2, NULL, print_message, msg2);
    pthread_create(&t3, NULL, print_message, msg3);
    pthread_create(&t4, NULL, print_message, msg4);

    //wait until the thread completes
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);

    return 0;
}
```

Explanation:
Four threads are created by this C program, and each one prints a distinct message along with its thread ID (TID) and process ID (PID). Each thread runs the print_message function, which outputs the supplied message, TID, and PID. pthread_join call makes sure that each thread finishes before the program ends, whereas pthread_create call initializes four threads with distinct messages in main function. The program indicates successful execution by returning 0.

5.1a

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

void* hello_fun()
{
    printf("Hello World!\n");
    return NULL;
}

int main(int argc, char* argv[])
{
    pthread_t thread;

    pthread_create(&thread, NULL, hello_fun, NULL);

    pthread_join(thread, NULL);

    return 0;
}
```

```
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ gcc list3.c -o list3 -lpthread
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ ./list3
Hello World!
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ nano list3.c
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$
```

5.1b

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

void* thread_func_exit(void* arg)
{
    printf("Thread with exit() - Exiting and killing the entire process.\n");
    exit(0);
}

void* thread_func_pthread_exit(void* arg)
{
    printf("Thread with pthread_exit() - Exiting this thread only.\n");
    pthread_exit(NULL);
}

int main()
{
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, thread_func_exit, NULL);
    sleep(1);
    pthread_create(&thread2, NULL, thread_func_pthread_exit, NULL);
    pthread_join(thread2, NULL);

    printf("Main thread: This message will only appear if exit() was not called.\n");

    return 0;
}
```

```
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ nano multi_thread_exit.c
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ gcc multi_thread_exit.c -o multi -lpthread
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ ./multi
Thread with exit() - Exiting and killing the entire process.
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$
```

Explanation:
To show the difference between exit and pthread_exit, this code generates two threads. The program is terminated and all threads are immediately stopped when the first thread executes a function that calls exit. In order to stop that particular thread and permit the remainder of the program to continue, the second thread executes a procedure that calls pthread_exit. While pthread_exit just stops the caller thread, exit terminates everything, as demonstrated by the use of a sleep to guarantee that the first thread runs first. The last printf statement in main won't run if exit runs.

6.1a

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <pthread.h>
#include <unistd.h>

void *function1(void *args) {
    int *a = (int *)args;
    for (int i = 0; i < 200000000; i++)
        a[i % 100000000] = 1000000000;
    return NULL;
}

void *function2(void *args) {
    int *a = (int *)args;
    for (int i = 200000000; i < 400000000; i++)
        a[i % 100000000] = 2000000000;
    return NULL;
}

void *function3(void *args) {
    int *a = (int *)args;
    for (int i = 400000000; i < 600000000; i++)
        a[i % 100000000] = 3000000000;
    return NULL;
}

void *function4(void *args) {
    int *a = (int *)args;
    for (int i = 600000000; i < 800000000; i++)
        a[i % 100000000] = 4000000000;
    return NULL;
}
```

```c
int main(int argc, char *argv[]) {
    int *a = malloc(1000000000 * sizeof(int));
    if (a == NULL) {
        perror("ERROR");
        exit(EXIT_FAILURE);
    }

    pthread_t t1, t2, t3, t4; // thread identifiers

    // Create threads to initialize parts of the array with different values
    pthread_create(&t1, NULL, function1, a);
    pthread_create(&t2, NULL, function2, a);
    pthread_create(&t3, NULL, function3, a);
    pthread_create(&t4, NULL, function4, a);

    // Wait for all threads to complete
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);

    free(a);
    exit(EXIT_SUCCESS);
}
```

Explanation:
Four functions are defined, each of which is run by a different thread. function1 till function4 use modulo indexing to fill parts of the array with 1,000,000,000, 2,000,000,000, 3,000,000,000, and 4,000,000,000, respectively. After creating the threads and allocating memory for the array, the main function waits for them to finish before releasing the memory and ending. This method shows how to efficiently fill a huge data structure using parallel processing.

6.1b

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

#define ITERATIONS 10000000

long long counter = 0;

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

int main(int argc, char *argv[]) {
    pthread_t t1, t2, t3, t4; // thread identifiers

    // Create threads
    pthread_create(&t1, NULL, increment_counter, NULL);
    pthread_create(&t2, NULL, increment_counter, NULL);
    pthread_create(&t3, NULL, increment_counter, NULL);
    pthread_create(&t4, NULL, increment_counter, NULL);

    // Join threads
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);

    printf("The value of the counter is %lld\n", counter);
    exit(EXIT_SUCCESS);
}
```

```
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ nano 6b.c
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ gcc 6b.c -o 6b -lpthread
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ ./6b
The value of the counter is 11726466
```

```
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ gcc 6b.c -o 6b -lpthread
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ ./6b
The value of the number is 14100559
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ nano 6b.c
```

Explanation:
I used multiple threads in this C program to increase a global counter variable. I defined a function named increment_counter that executes the increment in a loop, and I defined a constant ITERATIONS set to 10 million. I also used pthread_create to create four threads in the main function, each of which runs the increment function. After starting the threads, then pthread_join is used which waits for each one to finish before publishing the counter's final value. I should point out, though, that this approach is not synchronised, which may result in race situations when several threads try to access the counter at once.
Everytime when I rerun this program the value of counter changes

7

pthread_exit()

The pthread_exit(void *retval) function is used to end a thread's execution and can return a value to any thread that joins it. As long as there are other threads operating, the application will continue even when a thread that calls this function ends. It takes one parameter, retval, which is a pointer that can send back a value. It indicates that no value is returned if set to NULL. In order to offer a meaningful exit status, this method is frequently called at the conclusion of a thread.

```c
void* thread_func_pthread_exit(void* arg)
{
    printf("Thread with pthread_exit() - Exiting this thread only.\n");
    pthread_exit(NULL);
}

int main()
{
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, thread_func_exit, NULL);
    sleep(1);
    pthread_create(&thread2, NULL, thread_func_pthread_exit, NULL);
    pthread_join(thread2, NULL);

    printf("Main thread: This message will only appear if exit() was not called.\n");

    return 0;
}
```

pthread_self()

The pthread_self() function returns the identifier (ID) of the thread that calls it. When a thread needs to know its own ID for logging or particular operations, this is useful. It returns a value of type pthread_t, which is the thread's unique ID, and doesn't accept any parameters. We used this in exercise 3.1

```c
void* function4(void* arg)
{
    printf("Hello from Function4! Thread ID (TID): %lu, Process ID (PID): %d\n", pthread_self(), getpid());
    return NULL;
}

int main()
{
    //creating multiple threads
    pthread_t thread1, thread2, thread3, thread4;
    // create new threads and assign different functions to each
    pthread_create(&thread1, NULL, function1, NULL);
    pthread_create(&thread2, NULL, function2, NULL);
    pthread_create(&thread3, NULL, function3, NULL);
    pthread_create(&thread4, NULL, function4, NULL);
    // wait until each thread completes
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    pthread_join(thread4, NULL);

    printf("All threads have completed their tasks.\n");
    return 0;
}
[ Read 47 lines ]
```

pthread_equal()

The pthread_equal(pthread_t t1, pthread_t t2) function compares two thread IDs to see if they are the same. To determine whether two thread IDs are the same, the

pthread_equal(pthread_t t1, pthread_t t2) function compares them. If two parameters of type pthread_t indicate the same thread, it returns a non-zero result; otherwise, it returns zero.

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

void *thread_function(void *arg) {
    printf("Thread %ld is running.\n", pthread_self());
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    // Create the first thread
    pthread_create(&thread1, NULL, thread_function, NULL);
    // Create the second thread
    pthread_create(&thread2, NULL, thread_function, NULL);

    // Wait for both threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Compare the thread IDs
    if (pthread_equal(thread1, thread2)) {
        printf("Thread 1 and Thread 2 SAME.\n");
    } else {
        printf("Thread 1 and Thread 2 DIFFERENT.\n");
    }

    return 0;
}
```

```
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ nano pthread_equal.c
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ gcc pthread_equal.c -o equal -lpthread
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ ./equal
Thread 130491875329600 is running.
Thread 130491885815360 is running.
Thread 1 and Thread 2 DIFFERENT.
```

`pthread_detach()`

The pthread_detach(pthread_t thread) function is used to mark a thread so that its resources are automatically released when it finishes running. This implies that no additional thread is required to join the system when the thread is finished cleaning up. The ID of the thread to detach is the only parameter it requires. An error will appear if you attempt to disconnect a thread more than once or after it has been joined.

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

void *thread_function(void *arg) {
    printf("Hey there! I'm the detached thread!!\n");
    sleep(2);
    printf("All done here in the detached thread!\n");
    return NULL;
}

int main() {
    pthread_t thread; // Thread identifier

    // Create a new thread
    pthread_create(&thread, NULL, thread_function, NULL);

    // Detach the thread to allow it to clean up automatically when finished
    pthread_detach(thread);

    printf("Meanwhile, the main thread is busy with other tasks...\n");
    sleep(1);

    printf("Main thread wrapping things up now.\n");
    return 0;
}
```

```
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ nano pthread_detach.c
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ gcc pthread_detach.c -o detach -lpthread
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab11$ ./detach
Meanwhile, the main thread is busy with other tasks...
Hey there! I'm the detached thread!!
Main thread wrapping things up now.
```