POSIX THREADS (PTHREADS)

1. Threads are another mechanism for implementing concurrent programs that allow faster creation, reduced memory usage, and much simpler communication. POSIX threads (Pthreads) are the native threads implementation in Linux, but every modern operating system provides thread creation libraries. Below is a very basic thread creation example, using the Pthreads library.

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

void* f(void* a) {
    printf("f\n");
    return NULL;
}
```

```c
int main(int argc, char** argv) {
    pthread_t t;

    pthread_create(&t, NULL, f, NULL);
    printf("main\n");
    pthread_join(t, NULL);
    return 0;
}
```

   a. A thread always execute a given function, in our case it is f ( ).
   b. The Pthread library requires the thread function to have a specific signature. It should have a single **void*** argument that allows passing custom data to the thread and it should return **void*** that is useful for sending a result back to the main thread (via pthread_join). If f( ) returns something, you can receive it using a pointer pthread_join(t, &result); where result is void*.
   c. The call **pthread_create** creates and starts a thread that executes the given function pointer f. **&t** is a pointer to a pthread_t variable, which will be filled with the thread ID.
   d. The second argument to **pthread_create** is a pointer to **pthread_attr_t** which controls various aspects of the thread creation and execution such as stack size, scheduling policy, etc. In our case, by setting it to **NULL**, we use default settings.
   e. The fourth argument to **pthread_create** is the argument to be passed to function **f( )**. In our case, **f( )** doesn't use its argument, so we pass **NULL**.
   f. The call to **pthread_join** blocks the main thread until the thread identified by handle **t** end. Without pthread_join, the main thread might exit before the thread t finishes its task.
   g. The second argument to **pthread_join** will contain the value returned by function **f( )**. Passing it as **NULL** means we do not need this value, so ignore it.
   h. To compile a program that uses Pthreads, you need to either pass the -**pthread** argument to **gcc**, or to tell it to use the Pthreads library, which you can do passing the **-lpthread** argument.

```
gcc -Wall -Wextra -Werror -g -o a a.c -pthread
gcc -Wall -Wextra -Werror -g -o a a.c -lpthread
```

   i. The output of this program depends on how the operating system schedules the execution of these threads, consequently we may see the two **printf** value displayed in any order. This is essential to understanding how concurrency works.

2. It is difficult to see the non-deterministic aspects of thread scheduling using the program above, because the thread execution is extremely short, and it takes a lot of trials to see the lines printed in a different order.

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

int n = 1;

void* fa(void* a) {
    int i;
    for(i=0; i<n; i++) {
        printf("fa\n");
    }
    return NULL;
}

void* fb(void* a) {
    int i;
    for(i=0; i<n; i++) {
        printf("fb\n");
    }
    return NULL;
}
```

```c
int main(int argc, char** argv) {
    int i;
    pthread_t ta, tb;

    if(argc > 1) {
        sscanf(argv[1], "%d", &n);
    }

    pthread_create(&ta, NULL, fa, NULL);
    pthread_create(&tb, NULL, fb, NULL);

    for(i=0; i<n; i++) {
        printf("main\n");
    }

    pthread_join(ta, NULL);
    pthread_join(tb, NULL);

    return 0;
}
```

a. We now have two thread functions that is each run i a separate thread
b. The command line argument specifies how many iterations each thread will do. Notice that each thread has access to the global variable n
c. To show the thread scheduling without having long printouts, we pipe the output of the program through **uniq -c** without sorting the out first, thus displaying how many iterations each thread did before the scheduler switched to another. Be;low are two executions of the same program, taken a few moments apart.

```
./tb 10000|uniq -c
  201 main
    1 fa
   51 main
    1 fa
    1 main
    1 fa
   27 main
    1 fa
   30 main
    1 fa
 9690 main
10000 fb
 9995 fa
```

```
./tb 10000|uniq -c
 189 fa
   1 main
  50 fa
   2 main
   1 fa
   1 main
   1 fa
   1 main
  31 fa
   1 main
  31 fa
   1 main
  34 fa
   1 main
  35 fa
   1 main
 168 fa
   1 main
  11 fa
   1 main
```

```
   1 fa
  27 main
   1 fa
   1 main
  35 fa
   1 main
  34 fa
   1 main
   1 fa
  34 main
   1 fa
   1 main
   1 fa
  33 main
   1 fa
  36 main
   1 fa
9855 main
5153 fb
   1 fa
```

```
   1 fb
  15 fa
   1 fb
   1 fa
  34 fb
   1 fa
  36 fb
   1 fa
  36 fb
   1 fa
   1 fb
   1 fa
  35 fb
   1 fa
  34 fb
   1 fa
  37 fb
   1 fa
 115 fb
   1 fa
```

```
   1 fb
  28 fa
   1 fb
   1 fa
   1 fb
   2 fa
   1 fb
   1 fa
   1 fb
   1 fa
  31 fa
   1 fb
   1 fb
   1 fa
  35 fb
   1 fb
  34 fb
   1 fa
  34 fb
   1 fa
```

```
   1 fb
   2 fa
   1 fb
  31 fa
   1 fb
  32 fa
   1 fb
  35 fa
   1 fb
   1 fa
   1 fb
  35 fb
   1 fb
  37 fa
   1 fb
9107 fa
4466 fb
```

PTHREAD ARGUMENT PASSING

1. The last program can be written using a single thread function that gets value to display in the console as argument.

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

int n = 1;

void* f(void* a) {
    int i;
    for(i=0; i<n; i++) {
        printf("%s\n", (char*)a);
    }
    return NULL;
}

int main(int argc, char** argv) {
    int i;
    pthread_t ta, tb;
    if(argc > 1) {
        sscanf(argv[1], "%d", &n);
    }

    pthread_create(&ta, NULL, f, "fa");
    pthread_create(&tb, NULL, f, "fb");

    for(i=0; i<n; i++) {
        printf("main\n");
    }

    pthread_join(ta, NULL);
    pthread_join(tb, NULL);

    return 0;
}
```

2. If we want to pass n as an argument to the thread too (instead of having it a global variable), we need to declare a since the thread function can only get one argument.

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

struct arg_t {
    char* name;
    int count;
};

void* f(void* a) {
    int i;
    struct arg_t* x = (struct arg_t*)a;
    for(i=0; i<x->count; i++) {
        printf("%s\n", x->name);
    }
    return NULL;
}

int main(int argc, char** argv) {
    int i, n = 1;
    pthread_t ta, tb;
    struct arg_t aa, ab;
    if(argc > 1) {
        sscanf(argv[1], "%d", &n);
    }

    aa.name = "fa"; ab.name = "fb";
    aa.count = n; ab.count = n;

    pthread_create(&ta, NULL, f, &aa);
    pthread_create(&tb, NULL, f, &ab);

    for(i=0; i<n; i++) {
        printf("main\n");
    }

    pthread_join(ta, NULL);
    pthread_join(tb, NULL);

    return 0;
}
```

3. Let's create 10 threads, pass to each of them the order number in which it was created, and have it displayed. The program will include a serios bug that causes to a race condition and yields puzzling outputs.

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

void* f(void* a) {
    printf("%d\n", *((int*)a));
    return NULL;
}

int main(int argc, char** argv) {
    int i;
    pthread_t t[10];

    for(i=0; i<10; i++) {
        pthread_create(&t[i], NULL, f, &i);
    }

    for(i=0; i<10; i++) {
        pthread_join(t[i], NULL);
    }

    return 0;
}
```

a. Multiple executions of this program yielded the outputs below

A	B	C	D	E
1	1	1	1	0
2	5	5	2	1
6	2	3	6	1
6	6	4	6	1
3	6	2	3	1
4	6	7	4	1
7	7	6	7	1
8	8	8	8	1
9	9	4	8	1
4	4	5	8	1

b. The root cause of these behaviors is passing of &i to all threads. While we intend to pass to each thread the order number at which it was created, in reality, we pass to all threads the same value : the address of **i**. One of the 10nd thread might begin running when i has already changed, which is why you sometimes get outputs in the wrong or random order. The main function changes the value of **i** in two **for** cycles. Depending how the threads get the CPU, they will print whatever they find at the memory location **&i**.

f. The outputs of these implementations will still be non-deterministic although each thread receives the address of its own a[i] they can start executing before, after, or while the loop continues.

A	B	C	D	E
0	0	1	1	0
5	5	4	4	6
1	2	5	0	1
6	1	2	2	4
3	8	3	5	2
4	6	6	3	7
2	3	0	6	3
7	7	8	7	8
9	9	7	8	9
8	4	9	9	5