POSIX SYNCHRONIZATION MECHANISMS

1. Let's write a program that creates 10 threads, each of which increments a global variable as many times given as command line argument.

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

int count = 0;

void* f(void* a) {
    int i;
    for(i=0; i<*(int*)a; i++) {
        count++;
    }
    return NULL;
}

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

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

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

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

    printf("%d\n", count);

    return 0;
}
```

2. As we mentioned before race conditions do not always manifest, the best way to make them more likely to appear is to increase the concurrency. We'll run this program with various arguments and repeating those executions to see whether the output gets corrupted or not.

```
for n in 10 100 1000 10000 100000 1000000
do
    k=0
    while [ $k -lt 10 ]; do
        ./th $n
        k=`expr $k + 1`
    done
done
```

Correct result:	10	100	1000	10000	100000	1000000
	100	1000	10000	86337	610108	5068606
	100	1000	10000	74379	641742	5342264
	100	1000	9446	85734	590373	5777329
	100	1000	10000	82615	674556	5122867
	100	1000	9526	81122	601525	5477098
	100	1000	9754	79517	1000000	5288377
	100	1000	10000	79977	671355	5405598
	100	1000	10000	78419	599169	5340372
	100	1000	9115	81924	559812	5411218
	100	951	10000	77140	557539	2303521

3. You can see that as the concurrency increases, the values are more and more corrupted. Even though with lower concurrency things appear to be correct, it is just a matter of time until the data will be corrupted there as well.
4. This situation is called **race condition**, with count being the critical resource and the line **count++**; being the critical section.
5. Race conditions are solved using synchronization mechanisms.

MUTEXES

1. A mutex is a synchronization mechanism that has two main operations : **lock** and **unlock**.
2. Only one thread can complete the **lock** operation at one time, any other threads attempting to **lock** the mutex will have to wait until the "winning" thread calls **unlock**.
3. The race condition in the code above can be avoided using a mutex as shown below

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

int count = 0;
pthread_mutex_t m;

void* f(void* a) {
    int i;
    for(i=0; i<*(int*)a; i++) {
        pthread_mutex_lock(&m);
        count++;
        pthread_mutex_unlock(&m);
    }
    return NULL;
}

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

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

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

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

    printf("%d\n", count);

    return 0;
}
```

a. We wrap the critical section between mutex **lock**/**unlock** calls.
b. The **lock**/**unlock** operations could be moved outside the **for** loop. However, that would drastically reduce the concurrency, as the entire code of thread would be executed non concurrently.
c. The second argument to pthread_mutex_init is a pointer to pthread_mutexattr_t which controls various aspects of the mutex creation and execution. In out case, by setting it to NULL, we use the default settings.

READ-WRITE LOCKS

1. Consider a situation where the threads are acting on critical resource that can be grouped into readers (only consult the resource value without modifying it) and writers (modify the resource value). Also assume that the number of reader threads is significantly larger than that of writer threads.
2. Synchronizing the access to the critical resource with a mutex would prevent writers from corrupting the value, and readers from reading dirty values. However, the mutex will also prevent more than one reader from consulting the resource at one time, although any number of readers could consult it safely since they do not change it. So the mutex is too strict, it serializes all access for readers.
3. Read-write locks are designed to address this sittuation by providing two types of locking :
    a. **Write lock** : exclusive lock just like a mutex lock, only one thread can get a write lock at any time, but only if there are no read locks already present. An existing write lock will cause any read lock attempts to wait.
    b. **Read lock** : shared lock that can be obtained any number of times as long as there is no write lock present. The presence of any read locks will cause a write lock attempt to wait.
4. Here is an example simulating the purchasing product X of which there is an initial stock of 100 pieces. Most shoppers just check the availability of the product, and only some of them buy it.

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

int x = 100;
pthread_rwlock_t rwl;

void* shopper(void* a) {
    long id = (long)a;
    if(id % 10 == 0) { // buyer
        pthread_rwlock_wrlock(&rwl);
        if(x > 0) {
            x--;
            printf("%ld: bought %d\n", id, x+1);
        } else {
            printf("%ld: none left\n", id);
        }
        pthread_rwlock_unlock(&rwl);
    } else { //onlooker
        pthread_rwlock_rdlock(&rwl);
        printf("%ld: available %d\n", id, x);
        pthread_rwlock_unlock(&rwl);
    }
    return NULL;
}
```

```c
int main(int argc, char** argv) {
    int i;
    pthread_t t[200];

    pthread_rwlock_init(&rwl, NULL);

    for(i=0; i<200; i++) {
        pthread_create(&t[i],
                       NULL,
                       shopper,
                       (void*)(long)i);
    }

    for(i=0; i<200; i++) {
        pthread_join(t[i], NULL);
    }

    pthread_rwlock_destroy(&rwl);
    return 0;
}
```

But why do we need a lock if we're only reading? To avoid a problem called a "dirty read". Here is what could happen without rdlock :

- Thread A (a reader) starts reading x.
- At the same time, Thread B (a writer) modifies x.
- Thread A reads a value in the middle of the modification - the result is incorrect.

Type of Lock	Other reads allowed?	Other writes allowed?
rdlock (read)	✅ Yes	❌ No
wrlock (write)	❌ No	❌ No

CONDITIONAL VARIABLES

1. **Pthread_cond_t** is a condition variable, that is always used together with a mutex and allows one thread to wait for a specific condition to become true. Another thread must signal the condition once it becomes true.
2. What happens when you call **pthread_cond_wait**(&cond, &mutex)?
   a. The thread unlocks the associated mutex, allowing other threads to enter the critical section.
   b. The thread is then put to sleep, waiting for a signal on the condition variable.
3. What happens when you call **pthread_cond_signal**(&cond)?
   a. One thread that is currently waiting on the condition variable is woken up.
   b. Before the thread continue executing the code ather pthread_cond_wait, it mush re-lock the mutex.
4. Here is an example, where a car needs 40 units of fuel to start. Another thread fills the fuel tank in parts, and the car thread waits until there is enough fuel.

```c
pthread_mutex_t mutexFuel;
pthread_cond_t condFuel;
int fuel = 0;

void *fuel_filling(void *arg)
{
    for (int i = 0; i < 5; i++)
    {
        pthread_mutex_lock(&mutexFuel);
        fuel += 15;
        printf("Filled fuel... %d\n", fuel);
        pthread_mutex_unlock(&mutexFuel);
        pthread_cond_signal(&condFuel);
        sleep(1);
    }
}

void *car(void *arg)
{
    pthread_mutex_lock(&mutexFuel);
    while (fuel < 40)
    {
        printf("No fuel. Waiting...\n");
        pthread_cond_wait(&condFuel, &mutexFuel);
        // Equivalent to:
        // pthread_mutex_unlock(&mutexFuel);
        // wait for signal on condFuel
        // pthread_mutex_lock(&mutexFuel);
    }
    fuel -= 40;
    printf("Got fuel. Now left: %d\n", fuel);
    pthread_mutex_unlock(&mutexFuel);
}

int main(int argc, char *argv[])
{
    pthread_t th[2];
    pthread_mutex_init(&mutexFuel, NULL);
    pthread_cond_init(&condFuel, NULL);
    for (int i = 0; i < 2; i++)
    {
        if (i == 1)
        {
            if (pthread_create(&th[i], NULL, &fuel_filling, NULL) != 0)
            {
                perror("Failed to create thread");
            }
        }
        else
        {
            if (pthread_create(&th[i], NULL, &car, NULL) != 0)
            {
                perror("Failed to create thread");
            }
        }
    }

    for (int i = 0; i < 2; i++)
    {
        if (pthread_join(th[i], NULL) != 0)
        {
            perror("Failed to join thread");
         }
     }
     pthread_mutex_destroy(&mutexFuel);
     pthread_cond_destroy(&condFuel);
     return 0;
 }
```

SEMAPHORES

1. A **semaphore** is a synchronization mechanism used to control access to a shared resource by multiple threads. It maintains an internal counter that represents the number of available permits. A thread only proceed if the counter is greater than 0. Otherwise, it blocks until a permit is available.
2. What happens when you call **sem_wait**()?
   • call decrements the counter (aquires a permit)
3. What happens when you call **sem_post**()?
   • call increments the counter (releases a permit)
4. Here is an example that simulates a car wash station that has only 4 washing slots for cars. We have 10 cars (threads) and each one of them needs to wait for a free spot.

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

void* car_wash(void* arg) {
        sem_wait(&semaphore);
        printf("Car %d is being washed now...\n", *(int *)arg);
        int n = random( ) % 10;
        sleep(n);
        printf("Car %d is done being washed!\n", *(int*) arg);
        sem_post(&semaphore);

        free(arg);
        return NULL;
}

int main( ) {
        sem_init(&semaphore, 0, 4);
        pthread_t t[10];
        for (int i = 0; i < 10; i++) {
                int *p = (int *)malloc(sizeof(int));
                *p = i;
                pthread_create(&t[i], NULL, car_wash, p);
        }

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

        return 0;
}
```

BARRIERS

1. A **barrier** is a synchronization mechanism used to control a group of threads so that no thread continues until all threads have reached the same point.
2. We use barriers to ensure that all threads complete a task before moving on to the next step, the initialization is **pthread_barrier_init**(&barrier, NULL, 4) and it means that 4 threads must call pthread_barrier_wait() before any of them can proceed past that point.
3. What happens when you call **pthread_barrier_wait**(&barrier)?
   - blocks the calling thread until the specified number of threads have reached the barrier, then unblocks all of them simultaneously
4. Here is an example where 10 threads work in phases. All mush finish phase 1 before phase 2 begins.

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

pthread_barrier_t barrier;

void *routine(void *args)
{
    while (1)
    {
        printf("Waiting at the barrier...\n");
        sleep(1);
        pthread_barrier_wait(&barrier);
        printf("We passed the barrier\n");
        sleep(1);
    }
}

int main(int argc, char *argv[])
{
    pthread_t th[10];
    int i;
    pthread_barrier_init(&barrier, NULL, 10);
    for (i = 0; i < 10; i++)
    {
        if (pthread_create(&th[i], NULL, &routine, NULL) != 0)
        {
            perror("Failed to create thread");
        }
    }
    for (i = 0; i < 10; i++)
    {
        if (pthread_join(th[i], NULL) != 0)
        {
            perror("Failed to join thread");
        }
    }
    pthread_barrier_destroy(&barrier);
    return 0;
}
```