Zero2Hero - Multi Threaded

Understanding Processes vs. Threads

When you write a code eventually gets compiled and assembled and transformed into a binary blob that the interpreter of your operating system is able to execute. When it executes that code, the kernel underneath your OS takes a space for memory and the space for program into a memory. It also tracks certain statistic about program, like how much memory has it used, what file descriptors are open inside that program, what other things has the program does through system calls all this information is tracked in the structure known as process. Process is a single cube of execution in the scope of an operating system. And an operating system's entire job is to take multiple processes, be able to run them, be able to run them, multi-process, have multiple of them running at the same time and the processes are not aware of each other. Meaning process A and process B all think that they have access to the entire memory space of the OS. Form zero to the FFFFFF and also means that have unlimited time in CPU. Now, in reality, that's not actually happening. The process is getting halted temporarily and then the kernel is coming in and swapping into new process, then swapping it out and going back to the old one. But as far as the process is aware, it has access to its own memory and it has access to all of the time in the world that it needs. It runs independently, alone and no one else can touch it. Now the core crux of an operating system is that it prevents process A and process B from talking to each other without permission. All of the traffic that goes from process A and process B need to be routed through the kernel where the kernel can check does the process A have permission to talk to process B, is the memory talk process A is asking to C in process B, is that protected memory, is that even allowed, is that a user address and thing like that. This all are processes, the smallest atom that OS can handle.
Threads in light weight process that have access to another process. For example, having process A, when a process A tries to make a thread in kernel all the kernel does it takes a copy of a process A -> A_1 and A_1 executes but A_1 has the same memory mapping as process A. They overlay the same memory. Kernel is creating another process to do scheduling with, to track the status of, to track the file descriptors with access to the same memory.
Fork and Thread are one system call but difference between them is flag in thread clone that gives the process access to the parent memory map.
In terms of programming the thread is the light weight process that lives in the same memory.

Creating and Terminating Threads

https://docs.oracle.com/cd/E19120-01/open.solaris/816-5137/tlib-1/index.html
Next step is to dive into the creation and management of separate threads of execution within out applications. This is where `pthread_create` comes in.

The word multi-threading can be translated as multiple threads of control or multiple flows of control. While a traditional UNIX process contains a single thread of control, multi-threading separates a process into may execution threads. Each of these threads runs independently.
The Pthread API Library
Multi-threading Terms:

- Process > the unix environment such as file descriptor, user ID, and so on, created with the fork system call, which is set up to run a program.
- Thread > A sequence of instructions executed within the context of a process.
- Single-threaded > Restricts access to a single thread. Execution is through sequential processing, limited to one thread of control.
- Multi-threading > allows access to two or more threads. Execution occurs in more than one thread of control, using parallel or concurrent processing.
- Mutial exclusion lock > object used to lock and unlock access to shared data. Such objects are also known as mutex.
- Condition variables > Objects used to block threads until a change of state.
- Read-write locks > object used to allow multiple read-only access to shared data, but exclusive access for modification of that data.
- Counting semaphore > a memory-based synchronization mechanism in which a non-negative integer count is used to coordinate access by multiple threads to shared resources.
- Parallelism > a condition that arises when at least two threads are executing simultaneously.
- Concurrency > a condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a from of virtual parallelism.

Creating a simple Thread

The point of this code is to create 10 threads. To have our one process. We have process A, that's our starting parent process and have it spawn 10 threads.
So the way we do this is we create first an array of this P thread structures, so these P thread ts are a structure into the P thread header file that track information about the state of thread. And If you want to later do something to that thread, like close it or try to send it data via some kind of I/O, we have to address it via this p thread structure.

We create thread with function `pthread_create` to spawn A thread in C.

```
emilija > man pthread_create

The  pthread_create()  function starts a new thread in the calling process.  The new
       thread starts execution by invoking start_routine(); arg is passed as the sole argu-
       ment of start_routine().

int pthread_create(pthread_t *restrict thread,
                          const pthread_attr_t *restrict attr,
                          typeof(void *(void *)) *start_routine,
                          void *restrict arg);
```

Create a process and it is gonna write out all the information to that structure as pointed to at the first argument. So we give it an address of a thread, a P thread T, and inside the `pthread_attr` we can create an attribute structure that describe certain attributes about the process like CPU affinity, or how big we want the stack to be allowed to be. If we want to prevent the thread from allocating too much memory, we can do all of that in the attribute thing. But if we, put null here > pointer to zero and will give default attribute to parent. function start_routine which it takes the pointer to a function called start routine, and the idea begin that if we want information to be returned from P thread t, we can have that information come out from like a malloc buffer, for example. And that malloc buffer return value can come out of the function that we've specified. So when we wanna run a thread, we have to tell the thread what code to run and that code is going to be addressed by the function name. And last is argument, si this will be the address to our argument for our thread.
On success, `pthread_create()` return 0; on error, it returns an error number the contents of `*thread` are undefined.

```

```

```
#include <stdio.h>
#include <pthread.h>
#define THREAD_COUNT 10

void *thread_target(){
    printf("Hello, i am a thread\n")
}

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

    int i = 0;
    for(i = 0; i < THREAD_COUNT; i++){
        if(pthread_create(&threads[i], NULL, thread_target, NULL)){
            perror("pthread_create");
            return -1;
        }
    }

    return 0;
}

emilija@fedora:~/Documents/zero2_hero_c/multithreading$ ./creating-and-terminating-threads
Hello, i am a thread
Hello, i am a thread
```

```
Hello, i am a thread
Hello, i am a thread
Hello, i am a thread
Hello, i am a thread
```

`gcc -o creating-and-terminating-threads creating-and-terminating-threads.c -pthread` > using -pthread we tell the linker to pull in that data at link time.

We can see that the output is always different and running this program, and the p thread create thread, we are able to create the threads extremely fast, but we are not waiting for the threads to terminate execution. We are not waiting for the threads to what is called join. This will wait for all the threads to join to make sure that every thread is gotten to the end of its execution but effectively out program is terminating before every thread is created.

Joining and Detaching Threads

After initiating threads with `pthread_create`, it's essential to manage their termination properly. For efficient resource management and to ensure that threads complete their tasks, we use `pthread_join`. This function blocks the calling thread until the specified thread terminates.
To make a thread exit there are three, or the thread complete to do that.

- `thread_exit` > terminate calling thread
- `thread_detach` > function marks the thread identified by thread as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread,
- `thread_join` > function waits for the thread specified by thread to terminate. If that thread has already terminated, then `pthread_join` returns immediately. The thread specified by thread must be joinable. Returns `void *` to RET value. So when you make a thread to wait for its execution to end we must also wait it to join unless we detach it. Main difference between `thread_join` and `thread_detach` is that `detach` does not block > it just detaches, it backs off, and `join` will block until the thread is complete.

```
#include <stdio.h>
#include <pthread.h>
#define THREAD_COUNT 10

void *thread_target(void *arg){
    printf("Hello, i am a thread\n");
}

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

    int i = 0;
    for(i = 0; i < THREAD_COUNT; i++){
        if(pthread_create(&threads[i], NULL, thread_target, NULL)){
            // perror("pthread_create");
            return -1;
        }
    }

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

    return 0;
}

emilija@fedora:~/Documents/zero2_hero_c/multithreading$ gcc -o joining-and-detaching-threads joining-
and-detaching-threads.c -pthread
emilija@fedora:~/Documents/zero2_hero_c/multithreading$ ./joining-and-detaching-threads
```

```
Hello, i am a thread
Hello, i am a thread
Hello, i am a thread
Hello, i am a thread
Hello, i am a thread
Hello, i am a thread
Hello, i am a thread
Hello, i am a thread
Hello, i am a thread
Hello, i am a thread
emilija@fedora:~/Documents/zero2_hero_c/multithreading$
```

Passing arguments to Threads

When working with threads in C using the POSIX Threads (pthreads) library, a common requirement is to pass mutiple arguments to the threads. Since `pthread_create` only allow us to pass a single void pointer to the thread function, we can use a structure to encapsulate multiple arguments.

```
#include <stdio.h>
#include <pthread.h>
#define THREAD_COUNT 10

typedef struct{
    int arg1;
    short arg2;
} pthread_arg_t;

void *thread_target(void *vargs){
    pthread_arg_t *args = (pthread_arg_t*)vargs;
    printf("Hello, i am a thread %d\n", args->arg1); // reference to field to structure
}

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

    pthread_arg_t myargs;

    int i = 0;
    for(i = 0; i < THREAD_COUNT; i++){
        myargs.arg1 = i;

        if(pthread_create(&threads[i], NULL, thread_target, (void*)&myargs)){
            return -1;
        }
    }

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

    return 0;
}


emilija@fedora:~/Documents/zero2_hero_c/multithreading$ ./passing-arguments-to-threads
Hello, i am a thread 1
Hello, i am a thread 3
```

```
Hello, i am a thread 2
Hello, i am a thread 5
Hello, i am a thread 5
Hello, i am a thread 6
Hello, i am a thread 7
Hello, i am a thread 8
Hello, i am a thread 9
Hello, i am a thread 9
emilija@fedora:~/Documents/zero2_hero_c/multithreading$
```

What happens to data when multiple threads are accessing it at the same time. Because we passing argument to the same address and only ever in reality one copy of this, we're passing the same address and updating value in loop > **race condition**.

Passing Arguments to Threads

When working with threads in C using the POSIX Threads (pthreads) library, a common requirement is to pass multiple arguments to the threads. Since `pthread_create` only allows us to pass a single void pointer to the thread function, we can use a structure to encapsulate multiple arguments.

```
#include <stdio.h>
#include <pthread.h>
#define THREAD_COUNT 10

typedef struct{
    int arg1;
    short arg2;
} pthread_arg_t;

void *thread_target(void *vargs){
    pthread_arg_t *args = (pthread_arg_t*)vargs;
    printf("Hello, i am a thread arg1: %d arg2: %d\n", args->arg1, args->arg2); // reference to field to structure
    return NULL;
}

int main(int argc, char *argv){
    pthread_t pthreadID;
    pthread_arg_t myargs;
    myargs.arg1 = 200;
    myargs.arg2 = 12;

    pthread_create(&pthreadID, NULL, thread_target, (void*)&myargs);
    pthread_join(pthreadID, NULL);

    return 0;
}
```

`thread_targer` > the thread function that we pass to `pthread_create` needs to be designed to accept a single void pointer. Within the function, we cast this pointer back to our specific structure type to access the arguments.
With the structure and thread function defined, we can create a thread and pass our structured arguments to it.

Understanding Thread Synchronization

Thread synchronization is arguable the most important part of learning how to do asynchronous threaded programming.
The idea of thread synchronization is basically if you have thread A and thread B and they both need to do work on the same data, how do you make it so that your program does not allow thread A or thread B to corrupt the other data or to access it in a way that is out of order. But effectively, because threads are all running at the same time and the data is globally mutable, meaning anyone can touch anyone else's data in a thread context, you're able to create these things called race conditions were basically

the functionality of your program is just a function of who gets the data first and that make you program behaves incorrectly but it also makes it behave non deterministically and in a way that people could take advantage of, or it could just cause your programs to break.

```
#include <stdio.h>
#include <pthread.h>
#define THREAD_COUNT 10

int counter = 0;
void *thread_target(void *vargs){
// critical section
    for(int i = 0; i < 10000; i++){
        counter++;
    }
    printf("Counter: %d\n", counter);
}

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

    int i = 0;
    for(i = 0; i < THREAD_COUNT; i++){
        if(pthread_create(&threads[i], NULL, thread_target, NULL)){
            return -1;
        }
    }

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

    return 0;
}
emilija@fedora:~/Documents/zero2_hero_c/multithreading$ ./understanding-thread-synchorinization
Counter: 10000
Counter: 19456
Counter: 25725
Counter: 26966
Counter: 25676
Counter: 29358
Counter: 31300
Counter: 33590
Counter: 36563
Counter: 46563
emilija@fedora:~/Documents/zero2_hero_c/multithreading$
```

Every time the program runs you get a new value for the maximum count, and it never goes up to the value that it should.
The problem is we have no thread synchronization because counter is global variable. What is happening we have 10 threads that are all running at generally the same time and what they're doing is they're pulling down the value counter into a local register, and then they're updating it and they're putting it back. So they're taking it down, they're incrementing it, and they putting it back into the counter global variable. But because that is all happening at the same time, multiple threads are taking a copy of counter and then incrementing it and then putting it back so the increments are getting lost because threads are overwriting the other threads work.

```
int counter = 0;
void *thread_target(void *vargs){
// critical section
```

```
    for(int i = 0; i < 10000; i++){
        counter++;
    }
    printf("Counter: %d\n", counter);
}
```

In a critical section, we need to make sure that the data is confined to access by one thread or it's only able to be worked on by one particular worker. Because if we don't, we are basically allowing the program to run completely randomly. We have no way to predict how the program is going to behave.

Problem with Thread Synchronization

Synchronization challenges include:

- Deadlocks: Occur when two or more threads hold resources and wait for the other to release their resource.
- Starvation: Happens when a thread never gets resources it needs because other threads are given priority.
- Concurrency Overhead: Managing thread synchronization can lead to performance overhead due to context switching and resource management.

pthread Features for Synchronization

`pthread` offers several mechanisms to manage thread synchornization:

- Mutex: `pthread_mutex_t` provides mutual exclusion allowing only one thread to access a resource at a time.
- Condition Variables: `pthread_cond_t` helps threads wait for certain conditions to be true before proceeding.
- Semaphores and Spinlocks: `sem_t` and `pthread_spinlock_t` control access and coordinate thread actions effeciently.

Mutex

Mutex are one of the thing you can do using P thread library to enable critical section to be exclusive meaning that one thread can access it at a time. To force only one thread to take some kind of action on time.

There are some edge cases where the print is out of sync so that it doesn't actually see the print on the time.
Using multithreading programming is to use multiple thread of computation to perform some action. The problem we're creating here is we actually have a significant performance issue. We are actually running like 90% on performat while this program goes on. While one thread gets to pass and do the action, we have nine thread that are in starvation. Starvation is this idea in threaded programming where basically a thread, this compute power that you borrow from the kernel there sitting and waiting > they are blocked and sleep because they're waiting for some kind of synchronization object a mutex, a semaphore and a spinlock they're waiting for it to come back.

When a mutex is locked, it actually puts the thread to sleep and the kernel uses those compute resources to go and do other things. When it does this, the catch gets invalidated in that thread and now everytime a thread comes back online when it tries to access the memory in this process, it invokes a cache miss in every one of those slept threads. Correct way of using multithreading is when you create multiple threads they are all doing different kinds of computation on diff objects, and ideally there are as few critial sections as possible. There are as few memory pieces that are global and needed to be accessed by all the threads.

Semapthores

Mutex or mutual excluded object is meant to prevent critical section in your code. The main diffrence between mutex and semaphores in C is that when a mutex is holding lock on resource > only the thread that hold a lock on mutex can that mutex unlock.
Semaphores are synchronization tools used to manage access to shared resources or coordinate activities across multiple threads. In a publisher-subscriber model, semaphores can effectively signal events and manage the flow of information between publishers and subscribers.

`semaphores.c`

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

```
sem_t sem;
// publisher > post some work
void *signal_event(void *arg){
    printf("Doing something...\n");
    sleep(2);
    printf("Singal event completion!\n");
    sem_post(&sem);

    // return;
}
// other, wait for work
void *wait_for_event(void *arg){
    printf("Waiting for event...\n");
    sem_wait(&sem);
    // wait for publisher to publish event and then we
    // unlocking the semaphore locked resource
    printf("Event has been triggered!\n");

    // return;
}
int main(int argc, char *argv[]){
    pthread_t t1,t2;
    sem_init(&sem, 0,0);

    pthread_create(&t1, NULL, wait_for_event, NULL);
    pthread_create(&t2, NULL, signal_event, NULL);

    pthread_join(t1,NULL);
    pthread_join(t2, NULL);

    sem_destroy(&sem);
    return 0;
}
```

`semaphores_1.c`

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

sem_t sem;
char message[100]; // shared resource

void *publisher(void *arg){
    printf("Publishing message...\n");
    sprintf(message, "Hello.");
    sem_post(&sem);
    printf("Message published.\n");

    return NULL;
}
void *subscriber(void *arg){
    printf("Waiting for message...\n");
    sem_wait(&sem);
    printf("Received message: %s\n", message);

    return NULL;
}
int main(int argc, char *argv){
    pthread_t pub, sub;

    sem_init(&sem, 0,0);

    pthread_create(&pub, NULL, publisher, NULL);
    pthread_create(&sub, NULL, subscriber, NULL);

    pthread_join(pub, NULL);
    pthread_join(sub, NULL);

    sem_destroy(&sem);
    return 0;


}
```

Spin Locks

Spinlocks are a type of synchronization mechanism used in concurrent programming to protect shared resources. They are particularly effective in low-latency systems where threads are expected to wait for a short duration to acquire a lock.
When you use mutex when you get to a mutex lock and mutex is already taken, process will put the sleep by kernel. In the spin lock example, CPU thread when mutex is lock the CPU thread is spinning and using that memory till the lock is unlocked.
Spin lock is good to use when you need to wait only for small amount of time (millisecond) like individual clock cycle at a time because you're getting into this region of computing where isn't enough time for a context switch to happened where the CPU puts out process on the back burner and then waits for, and then allows another process to run. If you gonna spin lock a little longer then a half a second even a 10th of a second, recommended is to use mutex.

Info

perf-stat - Run a command and gather performance counter statistics
documentation: https://man7.org/linux/man-pages/man1/perf-stat.1.html

example using **spin-lock**

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

pthread_spinlock_t lock;
int counter = 0;

void *increment_counter(void *arg){
    for(int i = 0; i < 10000; i++){
        pthread_spin_lock(&lock);
        counter++;
        pthread_spin_unlock(&lock);
        printf("Counter incremented to %d\n", counter);
    }

    return NULL;
}

int main(int argc, char *argv){
    pthread_t t1,t2;
    pthread_spin_init(&lock, 0);

    pthread_create(&t1, NULL, increment_counter, NULL);
    pthread_create(&t2, NULL, increment_counter, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_spin_destroy(&lock);

    return 0;
}
```

output:

```
Counter incremented to 19978
Counter incremented to 19979
Counter incremented to 19980
Counter incremented to 19981
Counter incremented to 19982
Counter incremented to 19983
Counter incremented to 19984
Counter incremented to 19985
Counter incremented to 19986
Counter incremented to 19987
Counter incremented to 19988
Counter incremented to 19989
Counter incremented to 19990
Counter incremented to 19991
Counter incremented to 19992
Counter incremented to 19993
Counter incremented to 19994
Counter incremented to 19995
Counter incremented to 19996
Counter incremented to 19997
Counter incremented to 19998
Counter incremented to 19999
Counter incremented to 20000

 Performance counter stats for './spin-lock':

        > 55,150,382      task-clock:u                     #    1.268 CPUs utilized  > total CPU time
spent executing your process, user-space only, miliseconds, 1.268 CPU > means more then one core is
used

        >        0      context-switches:u               #    0.000 /sec > number of times the
scheduler descheduled your thread, threads never slep if is zero > spin lock
                 0      cpu-migrations:u                 #    0.000 /sec > times a thread moved between
CPUs, 0 means threads stayed pinned to their inital cores
                63      page-faults:u                    #    1.142 K/sec > minor page faults, no disk
I/O, stack setup, libc
        21,063,047      instructions:u                   #    0.75  insn per cycle > retaired CPU
intructions, ~21 milion instructions total
        27,909,861      cycles:u                         #    0.506 GHz > CPU clock cycles spent
executing intructions, ~28 milions cycles
```

```
         4,312,480      branches:u                       #   78.195 M/sec >       loops, ifs
           221,758      branch-misses:u                  #    5.14% of all branches >     loops, ifs

       0.043502107 seconds time elapsed

       0.010363000 seconds user
       0.044700000 seconds sys - Spin locks push work into kernel mechanisms indirectly


emilija@fedora:~/Documents/zero2_hero_c/multithreading$
```

---

example using **mutex**

```
 Performance counter stats for './mutex':

         3,006,796      task-clock:u                     #    0.904 CPUs utilized
                 0      context-switches:u               #    0.000 /sec
                 0      cpu-migrations:u                 #    0.000 /sec
                78      page-faults:u                    #   25.941 K/sec
         6,180,999      instructions:u                   #    0.84  insn per cycle
         7,350,142      cycles:u                         #    2.445 GHz
         1,038,481      branches:u                       #  345.378 M/sec
             3,597      branch-misses:u                  #    0.35% of all branches

       0.003326106 seconds time elapsed

       0.002636000 seconds user
       0.001334000 seconds sys


emilija@fedora:~/Documents/zero2_hero_c/multithreading$
```

Why spinlocks can have performance issues

Spinlocks can cause performance degradation when used improperly due to their nature of busy-waiting. If a thread holding a spinlock is pre-empted while holding it, other threads will waste CPU cycles spinning and checking the lock. This is particularly problematic in environments with high contention or on system where thread priorities cause the lock holder to be scheduled less frequently.
While pthread spinlock are useful for short-duration lock where the overhead of context switching is more costly then the time spent spinning, they should be used carefully. Their misuse can lead to high CPU consumption and poor scalability in multi-core systems.

Thread-Specific Data in C

The mutex and semaphore case is used to control access to data that is global and accessible to all threads. This chapter will explain how to make memory that is only accessible by one thread at a time.

When developing multithreaded application in C, managing data that is unique to each thread is crucial for maintaining state and avoiding conflicts. This is typically achived through two main techiques: Thread-Specific Data (TSD) using `pthread_key_create`, and Thread-Local Storage (TLS) using the `__thread` storage class specifier.

Thread-Specific Data (TSD)

Thread Specific data provides a way to allocate and manage data that is specific to each thread. This method is particularly useful when using libraries that do not support thread-local storage or when you need to perform cleanup operations on the data when a thread exits.

Using `pthread_key_create`

The `pthread_key_create` function creates a key that is globally available to all threads but maps to thread-specific data. Each thread can associate a thread-specific value with this key.

- Create a key: Use `pthread_key_create` to create a key. Optionally, specify a desctructor function to clean up the data when the thread exits.
- Set `tread-specific` data: Use `pthread_setspecific` to associate a specific value with the key for the calling thread.
- Get `thread_specific` data: Use `pthread_getspecific` to retrieve the data associated with the key for the calling thread.
- Delete the key: Use `pthread_key_delete` to free the key when it is no longer needed.

You create pthread key `pthread_key_t` > this key lives in global space but is reference that every thread individually can access it and use it to index into its thread specific data and get access to specific variable.

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

pthread_key_t key;
void array_destructor(void* arr){
    free(arr);
    printf("Array freed for a thread\n");
}

void* thread_function(void* arg){
    int* my_array = (int*)malloc(10 * sizeof(int));

    pthread_setspecific(key, my_array);

    for(int i = 0; i < 10; i++){
        my_array[i] = i;
    }

    for(int i = 0; i < 10; i++){
        printf(" %d \n", my_array[i]);
    }

    printf("\n");
    return NULL;
}

int main(){
    pthread_t thread1, thread2;

    pthread_key_create(&thread1, NULL, thread_function, NULL);
    pthread_key_create(&thread2, NULL, thread_function, NULL);

    pthread_join(&thread1, NULL);
    pthread_join(&thread2, NULL);

    pthread_key_delete(key);

    return 0;
}
```

What it does it `malloc` from the heap some data `my_array` and we gonna use this key to be reference to `my_array`.
So if we want to get the `my_array` variable all we have to do is to say `pthread_setspecific(key, my_array)`, get specific for

that key ad it'll return the `my_array` pointer. By doing this, we are creating a way for, by referencing this key, we are getting access excursively to this `my_array` variable, and again it is out copy of it, no one can touch it unless we pass that pointer through some kind of information sharing scheme. So if we want to create that key, we use the key create, we provide the constructor, we run our functions, and once the threads end we run the key delete function and it will run the deconstruct function that we create to free memory space of array.

Thread-Local Storage (TLS)

Thread-Local Storage allows variables to be a local to a tread. Unlike TSD, TLS is a compiler feature, simplifying syntax and improving performance for accessing thread-local data.

Using `__thread`

The `__thread` specifier can be used to declare variables that are automatically thread-local.

- Declare a TLS variable: Prefix the variable declaration with `__thread`
- Access the TLS variable: Access and modify the variable as you would any other static or global variable, but it will be unique per thread.

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

__thread int x_thd;
void wait_for_event(){
    x_thd++;
    printf("%d\n",x_thd);
    return;
}

int main(){
    pthread_t threads[10];
    for(int i = 0; i < 10; i++){
        pthread_create(&threads[i], NULL, wait_for_event, NULL);
    }

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

    return 0;
}
```

When we have to create the key and then set the key to a certain variable and set up the deconstruction, there's actually a keyword in GCC called thread and what can we do with thread is it literally creates a global variable where instead of it being a global variable that everyone can touch, it is a global variable that is unique to an individual thread. The primary difference between these two is that the thread keyword is handled at compile time, whereas the thread specific data is actually handled at runtime and it's a lot heavier. The TSD, the previous method uses memory managers it has a lot to do with underlying post API whereas this is actually thread local storage it is handled in compile time. You can run n number times, and this data is independent for this thread, there is not global variable and there is not race condition.

Both TSD and TLS are powerful techniques for managing data specific to threads in a C program. While TSD provides more control and is more portable across different environments, TLS offers simplicity and performance benefits. Choose the technique that best fits your needs based on the constraints and requirements of your application.

Managing Thread Attributes

`pthread_attr_t` is a data structure in POSIX threads used to specify thread attributes when creating new threads. These attributes include detach state, stack size, scheduling policy, and more, allowing developers to control various aspects of thread behavior.
Setting stack size we're able to limit the task size of a P thread > if you want to be sure that thread does run off and use a bunch of memory, we wanna to limit how much stack space a thread can have. What we can do is we can actually use the set the stack size, say that three times fast.

```
# init a attribute
pthread_attr_t attr;
pthread_attr_init(&attr);

# initialize size of stack
size_t stacksize = 1024*1024;

# set stack size of attribute
pthread_attr_setstacksize($attr, stacksize);
```

We can also set a scheduling policy, we're using the same attribute structure and we gonna use scheduling for round robin (FIFO) of the thread. If you have 10 threads, none of them have priority and if we set scheduling policy, then execution of threads if first, second and ... 10th. With setting scheduling policy, we can also set the priority of particular thread to run at max priority.

```
pthread_attr_setschedpolicy(&attr, SCHED_RR);
struct sched_param param;

param.sched_priority = sched_get_priority_max(SCHED_PR);
pptherad_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
```

The CPU affinity, when you create a pthread or just a thread in general or really any CPU process, the CPU core that it goes on is a function of the operating system. The OS will tell the thread or the, OS will decide what core that process goes on. Now if you have specific needs where you're doing a lot of data processing for example. You want that data processing to happens on one particular core because all the data processing, if you go from one core to another, you'll start having a lot of cache misses because the cache is populated on one CPU core but not another. So what you can do is you can tell the OS i would like to run this thread on only this named CPU core. So what you do is you create a `cpu_set_t` with `CPU_ZERO(&cpuset)` and then you use a CPU set function to set `CPU_SET(0, &cupiset)` > i wan to be on core zero only and then use the attribute set affinity function to put that CPU set into the attributes of our thread.

You have to compile with GNU source defined so that it gets access to the CPU set functions of the CPU set macros and then you also have to run this code as root, because only root has the ability to change the scheduling and the affinity of code.
`emilija@fedora:~/Documents/zero2_hero_c/multithreading$ gcc -o managing-thread-attributes managing-thread-attributes.c -lpthread -D_GNU_SOURCE`

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

#define _GNU_SOURCE
#include <sched.h>
#include <errno.h>

void* thread_function(void* arg){
    pthread_t tid = pthread_self(); // obatain ID of the calling thread
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);

    if(pthread_getaffinity_np(tid, sizeof(cpu_set_t),&cpuset) == 0){
        printf("Thread %lu running on CPUs: ", (unsigned long)tid);
        for(int i = 0; i < CPU_SETSIZE;i++){
```

```c
            if(CPU_ISSET(i, &cpuset)){
                printf("%d ",i);
            }
        }
        printf("\n");
    }

    return NULL;
}

int main(){
    pthread_t thread;
    pthread_attr_t attr;

    cpu_set_t cpuset;
    struct sched_param param;

    pthread_attr_init(&attr);

    pthread_create(&thread, &attr, thread_function, NULL);
    size_t stacksize = 1024*1024;

    pthread_attr_setstacksize(&attr, stacksize);
    pthread_attr_setschedpolicy(&attr, SCHED_RR);

    param.sched_priority = sched_get_priority_max(SCHED_RR);
    pthread_attr_setschedparam(&attr, &param);
    pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);

    CPU_ZERO(&cpuset);
    CPU_SET(0, &cpuset);
    pthread_attr_setaffinity_np(&attr, CPU_SETSIZE, &cpuset);

    if(pthread_create(&thread, &attr, thread_function, NULL) != 0){
        perror("Failed to create thread");
        return 1;
    }

    pthread_join(thread, NULL);
    pthread_attr_destroy(&attr);
    return 0;
}
```

Using `pthread_attr_t` effectively allows for fine-tuned control over thread execution characteristic, enchaining the performance and predictability of multi-threaded application.Understanding how to manipulate these attributes, especially CPU affinity, is crucial for developing optimized parallel application.

Deadlock

Deadlocks occur in concurrent programming when two or more processes or threads are unable to proceed because each is waiting for the other to release a resource. This situation creates a cycle of dependencies that leads to a standstill, where none of the involved processes can continue execution. Deadlocks are particularly problematic in systems where multiple threads or processes must coordinate access to shared resources.
The Dining Philosophers problem is a classic example used to illustrate deadlocks. In this problem, philosophers sit around a table with a form between each pair. Each philosopher needs two forks to eat but can only pick up one fork at a time. The problem demonstrates how a circular wait condition can lead to a deadlock.

Dining Philosophers

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#define NUM_PHILOSOPHERS 5

pthread_mutex_t forks[NUM_PHILOSOPHERS];

void *philosopher(void* num){
    int id = *(int*)num; // philosopher id
    int left = id;
    int right = (id+1)%NUM_PHILOSOPHERS;

    while(1){
        printf("Philosopher %d is thiniking\n", id);
        sleep(1);
        printf("Philosopher %d is hungry\n", id);

        // try to pick up left fork
        pthread_mutex_lock(&forks[left]);
        printf("Philosopher %d picked up left fork %d\n", id, left);

        // try to pick up the right fork
        pthread_mutex_lock(&forks[right]);
        printf("Philosopher %d picked up right fork %d\n",id, right);

        printf("Philosopher %d is eating\n",id);
        sleep(1);

        // put down the right fork
        pthread_mutex_unlock(&forks[right]);
        printf("Philosopher %d put down right fork %d\n", id, right);

        // put down the left fork
        pthread_mutex_unlock(&forks[left]);
        printf("Philosopher %d put down left fork %d\n", id, left);
    }
}
int main(){
    pthread_t philosophers[NUM_PHILOSOPHERS];

    int ids[NUM_PHILOSOPHERS];

    // initialize forks > mutex
    for(int i = 0; i < NUM_PHILOSOPHERS; i++){
        pthread_mutex_init(&forks[i], NULL);
    }

    // init philosophers > threads
    for(int i = 0; i < NUM_PHILOSOPHERS; i++){
        ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &ids[i]);
    }

    // wait for all philosophers to finish (in this case, they never will)
    for(int i = 0; i < NUM_PHILOSOPHERS; i++){
        pthread_join(philosophers[i], NULL);
    }

    // destroy forks > mutex
    for(int i = 0; i < NUM_PHILOSOPHERS; i++){
        pthread_mutex_destroy(&forks[i]);
    }

    return 0;
}
```

1. **Philosopher Threads** > each philosopher is represented by a thread. The philosophers alternately think and try to eat;
2. **Forks as mutex** > the forks are represented by mutex. Philosophers must lock the fork mutexes to pick them up;
3. **Deadlock scenario** > a deadlock can occur if each philosopher picks up the left fork (lock the left mutex) and then waits indefinitely for the right fork, which is held by another philosopher. This creates a circular wait condition, leading to deadlock where all philosophers are waiting and none can proceed to eat.

Solutions to Prevent Deadlocks in the Dining Philosophers Problem

1. **Limit the Number of Accesses** (circular wait)
   One effective approach to preventing deadlocks is to **limit** the number of philosophers (or **processes**) **that can attempt to pick up resources (forks) at the same time.** This method works by ensuring that not all philosophers can become hungry and try pick up forks simultaneously, **reducing the chances of creating a circular wait.**
2. **Resource Hierarchy** (wait and hold)
   The resource hierarchy solution involves **assigning a strict order to the resources (forks) and requiring that all processes (philosophers) acquire resources in a specific order.** This prevents circular wait conditions, which are a key component of deadlocks.
3. **Non-blocking Techniques**
   Non-blocking techniques involve designing the system so that processes (philosophers) **do not wait indefinitely for resources**. If a philosopher **cannot acquire both forks, they release any resource they have already acquire and try again later.**

Multithreaded Applications

- coding > thread is not hard. Designing API is straightforward, make a tread, make a mutex, lock the thread , free the mutex.
- designing > threaded applications is hard, making use of the now parallelized code so that it doesn't lock itself and you're actually making use of all the resources in a way that is actually usefull.

Deadlock

A situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.
Strategies:

- Limit number of accesing
- Resource hierarchy
- Non-blocking

Performance Considerations

When developing multi-threaded application in C, performance is a critical factor. One significant aspect to consider is the overhead associated with creating and deleting threads. While threading can enhance performance by enabling parallel execution, inefficient management of threads can lead to overhead, diminishing the benefits.

Overhead of Creating and Deleting Threads

Creating and deleting thread in C program involves system calls that can be expensive in terms of CPU time and memory usage. Each time a thread is created, the operating system allocates resources such as memory for the thread's stack and data structures for managing the thread. When a thread is deleted, the OS must clean up these resources, which also incurs a cost.

- Thread Creation Cost > every time you create a new thread, the OS allocates memory and initializes the necessary data structures. This process can be time consuming if threads are created frequently,
- Thread Deletion Cost > deleting a thread also requires the OS to deallocate resources, which adds to the overall overhead. Frequent creation and deletion of threads can lead to performance bottlenecks, especially in applications that require high responsiveness or handle many short-lived tasks.

Using a Thread Pool to Improve Performance

A thread pool is a collection of pre-initialized threads that are keep alive and reused for execution tasks. Instead of creating and destroying a thread for each task, you can simply assign the task to an existing thread in the pool. This approach significantly reduces the overhead associated with thread management.

Advantages of a Thread Pool:

- Reduced Overhead > by reusing threads, you avoid the constant cost of creating and destroying threads, leading to more efficient CPU and memory usage.
- Improved Responsiveness > Task can be executed more quicky since threads are available and do not need to be created on demand.
- Scalability > thread pools make it easier to manage a large number of concurrent tasks, as you can control the maximum number of threads running at any given time.

Choosing where to use Threads

Deciding where to use threads in your application is crucial for optimizing performance. Not all tasks benefit from parallel execution, and threading introduces complexity that can sometimes outweigh the benefits.

Considerations for using Threads:

- **Task Granularity** > use threads for tasks that are sufficiently large or time-consuming. If a task is too small, the overhead of managing thread may negate any performance gains.
- **Concurrency Needs** > threads are beneficial when tasks can be performed concurrently without depending heavily on each other. Independent task are ideal candidates for threading.
- **Resource Contention** > Avoid using threads when tasks heavily contend for the same resources, as this can lead to performance degradation due to context switching and synchronization overhead.
- **System Resources** > consider the available CPU cores and memory. Creating too many threads can lead to resource exhaustion and decreased performance due to excessive context switching.