Analysis of Unreal Rendering System (02) - Multithread Rendering

Table of contents

2.1 Basics of Multithreaded Programming

In order to make a smoother transition, before actually entering the knowledge of UE's multithreaded rendering, first learn or review the basics of multi-threaded programming.

2.1.1 Multithreading Overview

The idea of multithreadedprogramming appeared as early as the single-core era. The operating system at that time (such as Windows 95) already supported multitasking. The principle was to switch different contexts in a single core so that threads in each process would have time to get the opportunity to execute instructions.

But by 2005, when the single-core clock speed approached 4GHz, CPU hardware manufacturers Intel and AMD discovered that speed would also reach its own limits: that is, simply increasing the clock speed could no longer significantly improve the overall performance of the system.

As the single-core computing frequency of Moore's Law slowly came to an end, Intel took the lead in launching the Pentium D and Pentium 4 Extreme Edition 840 series in 2005, which supported two physical-level thread computing units for the first time. In the more than ten years since then, multicore CPUs have flourished, and the Ryzen 3990X processor manufactured by AMD already has 64 cores and 128 logical threads.

The number of cores and threads is prominently displayed in the promotional poster for the Ryzen 3990X.

The development of multi-core hardware has given software a lot of room to play. Applications can fully utilize the computing resources of multi-cores and multi-threads, and various application fields have also produced multi-threaded programming models and technologies. Commercial engines Such as Unreal Engine, which is the engine of games, can also use multi-threading technology to further improve efficiency and effectiveness.

The effects of using multithreaded concurrency can be summarized into two main points:

- **Separation of concerns.**By separating related code from irrelevant code, the program can be easier to understand and test, thus reducing the possibility of errors. For example, in game engines, file loading and network transmission are usually placed in independent threads, which can not only not block the main thread, but also separate the logic code, making it
  clearer and more scalable.

- **Improve performance.**Strength lies in numbers, and this principle also applies to CPUs (more cores means more strength). If tasks of the same magnitude can be distributed to multiple CPUs and run simultaneously, efficiency will inevitably be improved.

However, as the number of CPU cores increases, the benefits of computers do not increase in a linear manner, but follow**Amdahl's law**.The formula of Amdahl's law is defined as follows:

$$Slatandncand(s) = \frac{1}{(1 \_ p) + p \frac{}{s}}$$

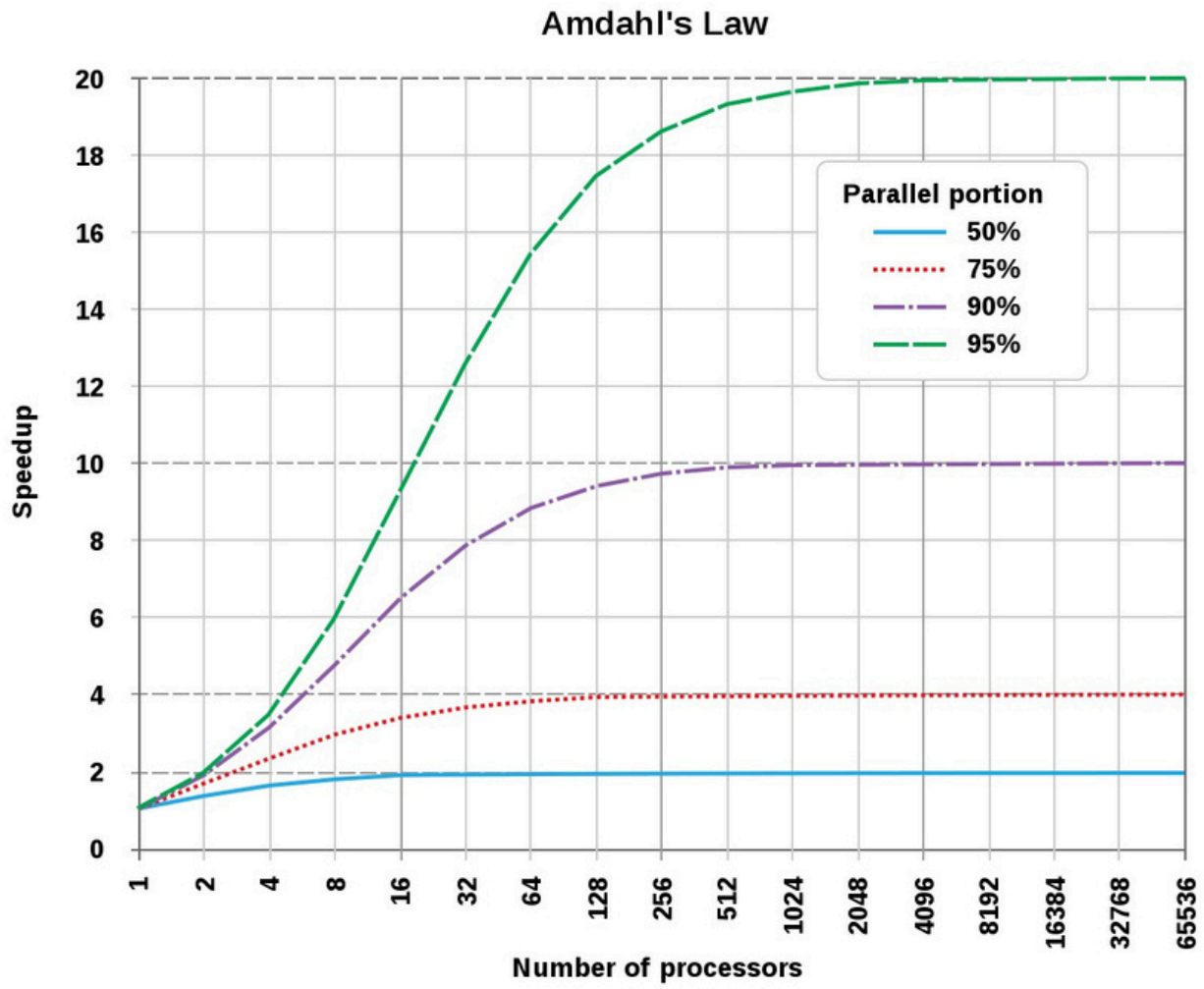
The meaning of each component of the formula is as follows:

- *Slatandncand*: The theoretical speedup ratio of the entire task in multi-threaded proce
- *s*: The number of threads of hardware resources used to execute the parallel part of t
- *p*: The proportion of tasks that can be processed in parallel.

For example, if there is an 8-core 16-thread CPU for a task, and 70% of the task can be processed in parallel, then its theoretical speedup is:

$$Slatandncand(16) = \frac{1}{(1 \_ 0.7) + \frac{0.7}{16}} = 2.9$$

It can be seen that the benefits of multithreaded programming are not directly proportional to the number of cores. In fact, its curve is as follows:



The core number and speedup ratio diagram revealed by Amdahl's law. It can be seen that the lower the proportion of parallelizable tasks, the worse the speedup ratio is: when the proportion of parallelizable tasks is 50%, 16 cores have basically reached the speedup ratio ceiling, no matter how many cores are added later, it will not help; if the proportion of parallelizable tasks is 95%, the speedup ratio ceiling will be reached when 2048 cores are used.

Although Amdahl's law brings us a cruel reality, if we can increase the task parallelism ratio to nearly 100%, the speedup ceiling can be greatly improved:

$$Slatandncand(s) = \frac{1}{(1\_p) + p\ \frac{}{s}} = \frac{1}{(1\_1) + s\ \frac{1}{}} = s$$

As shown in the above formula, when $p = 1$ (ie when the parallelizable tasks account for 100%), the theoretical speedup ratio is linearly proportional to the number of cores! !

For example, when compiling Unreal Engine project source code or Shader, since they are basically 100% parallel, theoretically a near linear acceleration ratio can be achieved, which will greatly shorten the compilation time in a multi-core system.

There are two ways to use multithreaded concurrency to improve performance:

- **Task parallelism.**Split a single task into several parts and run them in parallel to reduce the total running time. Although this method seems simple and intuitive, it may be complicated in

practice because there may be dependencies between the parts.

- **Data parallelism.**Task parallelism is the algorithm (instruction execution) part, that is, each thread executes different instructions; while data parallelism is the same instruction, but the data executed is different. SIMD is also a form of data parallelism.

The above explains the benefits of multithreaded concurrency. Next, let's talk about its side effects. In summary, the side effects are as follows:

- **Data contention.**Multithreaded access often cross-executes the same code, or operates the same resource, or there are high cache synchronization issues in multi-core CPUs. This change leads to various data asynchronization or data read and write errors, resulting in various abnormal results. This is data contention.
- **The logic is complicated and difficult to debug.**Since the concurrency mode of multithreading is not unique and unpredictable, complex and diverse synchronization operations are often added to avoid data competition. The code will become discrete, fragmented, cumbersome, and difficult to understand. Adding code assistance will bring immeasurable obstacles to subsequent maintenance and expansion. It will also cause bugs that are difficult to reproduce with low probability events, which increases the difficulty of debugging and error checking by an order of magnitude.
- **It may not necessarily improve efficiency.**Multithreading technology can indeed improve efficiency, but it is not absolute. It is often related to factors such as physical cores, synchronization mechanisms, runtime status, concurrency ratio, etc. In some extreme cases, or if it is not used properly, it may reduce program efficiency.

2.1.2 Multithreading Concept

This section will explain the basic concepts commonly involved in multi-threaded programming technology.

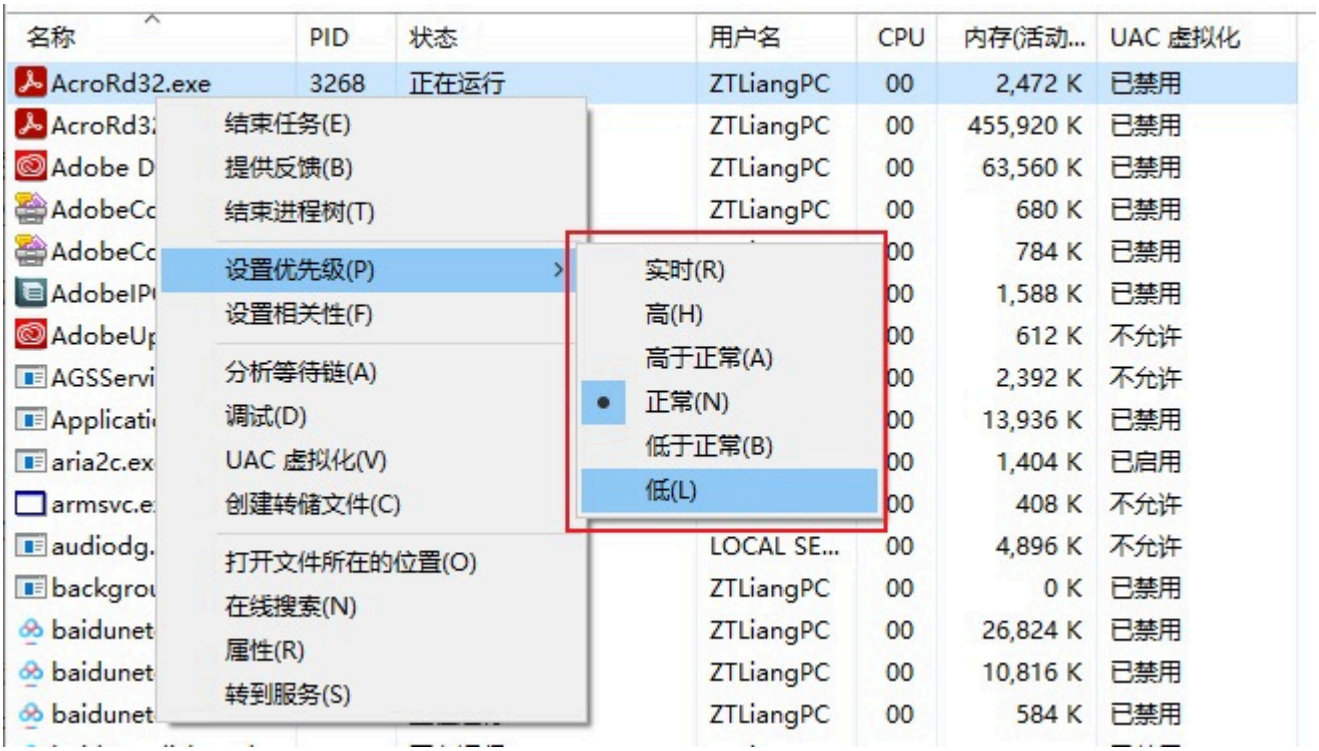
- **Process**

A processis the basic unit and entity for the operating system to execute applications. It is just a container, usually containing kernel objects, address space, statistics, and several threads. It does not actually execute code instructions, but rather leaves them to the threads within the process.

For Windows, when the operating system creates a process, it also creates a thread for it. This thread is called**the primary thread**(Main thread).

> For Unix, processes and main threads are actually the same thing. The operating system is unaware of the existence of threads. Threads are closer to the concept of lightweight processes.

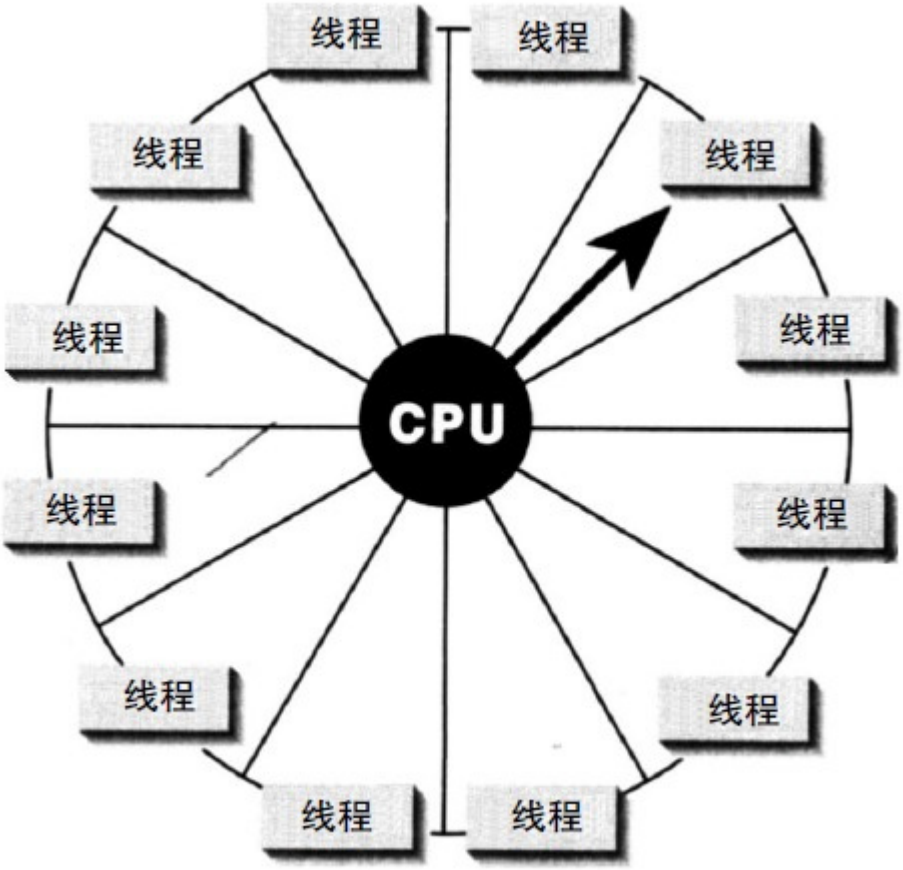
Processes have a priority concept, from low to high in Windows: Low, Below normal, Normal, Above normal, High, Real time. (See the figure below)



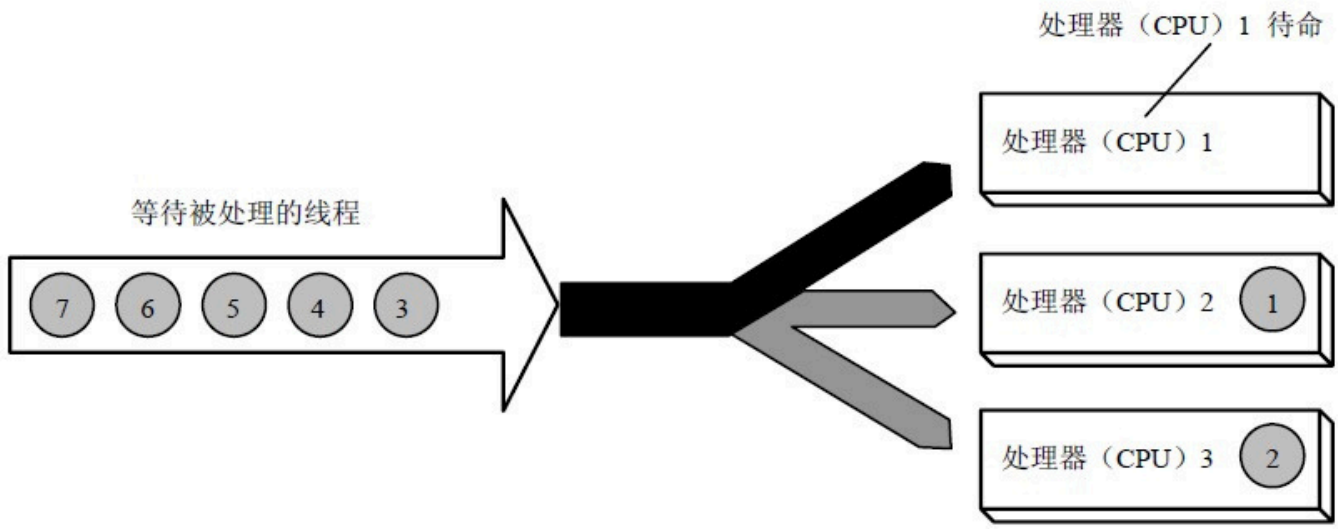
By default, the priority of a process is Normal. Processes with higher priorities will be given priority in execution opportunities and time.

- **Thread**

A threadis an entity that can execute code. It usually cannot exist independently and needs to be attached to a process. A process can have multiple threads, which can share the process's data to execute multiple tasks in parallel or concurrently. In a single-core CPU, the operating system (such as Windows) may use round-robin scheduling to
make multiple threads appear to run simultaneously. (Figure below)



In a multi-core CPU, threads may be scheduled to run simultaneously on different CPU cores, thereby achieving parallel processing.



Schematic diagram of Windows execution on a multi-core CPU using SMP. Threads waiting to be processed are scheduled to different CPU cores.

Each thread can have its own execution instruction context (such as Windows' IP (instruction register address) and SP (stack start register address)), execution stack and TLS (Thread Local Storage).

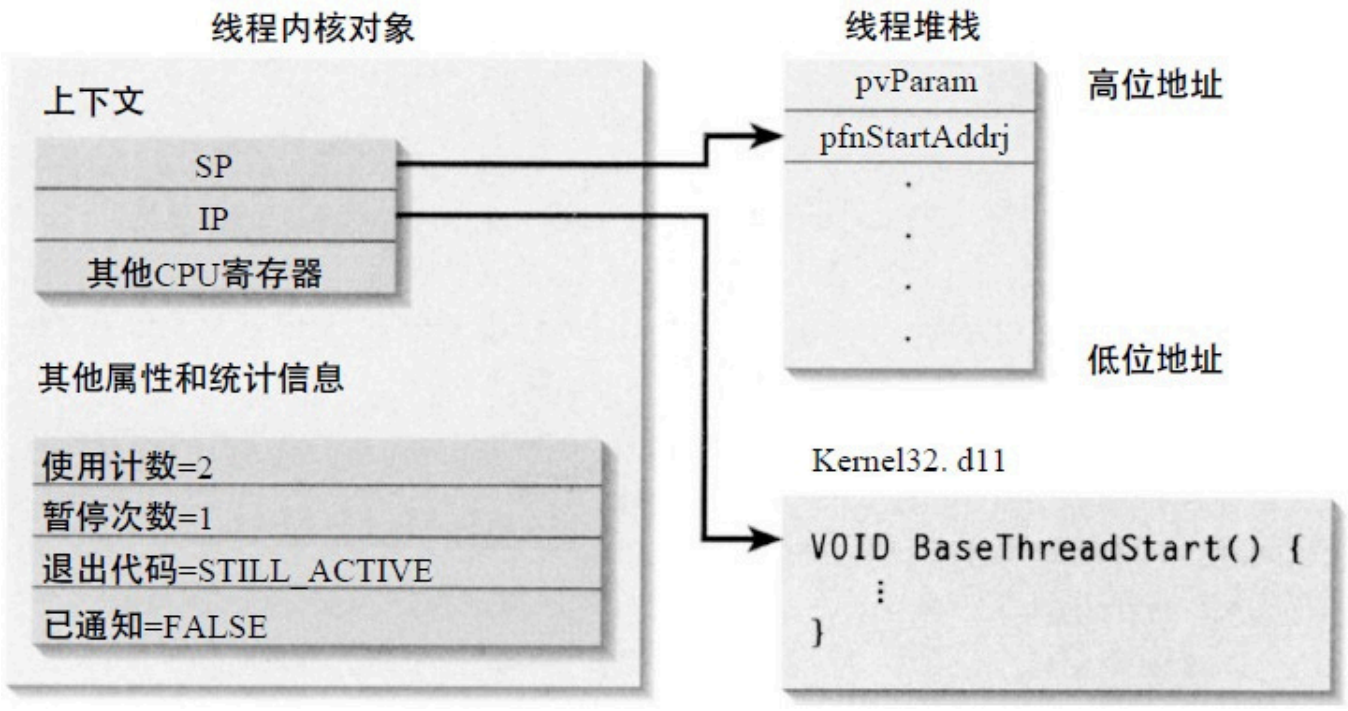


Diagram of Windows thread creation and initialization.

> **Thread Local Storage**is a storage duration. The life cycle of the object is the same as that of the thread. It is allocated when the thread starts and is reclaimed when the thread ends. Each thread has its own instance of the object. Accessing and modifying such an object will not cause race conditions.

Threads also have a priority concept, and threads with higher priorities have the opportunity to execute instructions first.

The thread status generally includes running status, paused status, etc. Windows can use the following interface to switch the thread status:

```
//Pause a thread
DWORDSuspendThread(HANDLE //Continue    hThread);
running the thread
DWORDResumeThread(HANDLE hThread);
```

The same thread can be suspended multiple times. If you want to resume the running state, you need to call the continue running interface the same number of times.

- **Coroutine**

Coroutineis a lightweight user-mode thread that usually runs on the same thread and uses different time slices of the same thread to execute instructions without the overhead of thread and process switching and scheduling. From the user's perspective, the coroutine mechanism can be used to simulate asynchronous tasks and coding methods on the same thread. In the same thread, it will not cause data competition, but it will also be blocked due to thread blocking.

- **Fiber**

Fibers,like coroutines, are also lightweight user-mode threads that allow applications to independently decide how their own threads should operate. The operating system kernel is unaware of the existence of fibers and will not schedule them.

- **Race Condition**

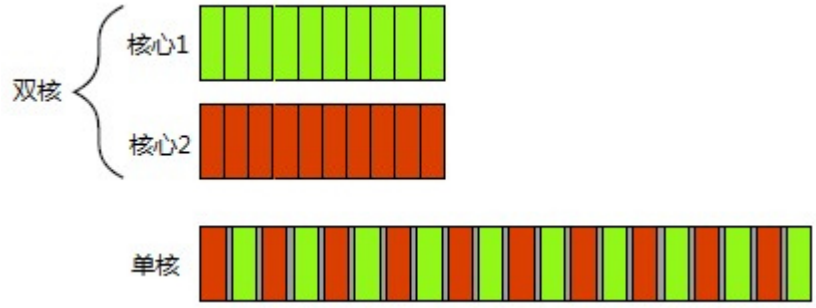
The same process allows multiple threads, which can share the process's address space, data structure, and context. Multiple threads may read and write the same data block in a process at the same time within a very small time segment, which will cause data anomalies and lead to unpredictable results. This unpredictability creates a**race condition**.
There are many techniques to avoid race conditions, such as atomic operations, critical sections, read-write locks, kernel objects, semaphores, mutexes, fences, barriers, events, and so on.

- **Parallelism**

A mechanism whereby at least two threads execute tasks simultaneously. Generally, only when multiple cores and multiple physical threads of a CPU execute tasks simultaneously can it be called parallelism, and multiple threads on a single core cannot be called parallelism.

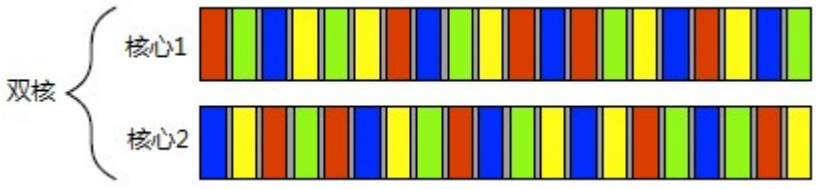
- **Concurrency**

The mechanism of at least two threads using time slices to execute tasks is a more general form of parallelism. Even multiple threads executed simultaneously by a single-core CPU can be called concurrent.



Two forms of concurrency - top: simultaneous execution of two physical cores (parallelism); bottom: multi-tasking switching of a single core (concurrency).

In fact, concurrency and parallelism can exist simultaneously in a multi-core processor. For example, as shown in the figure below, there are two cores, and each core switches multiple tasks at the same time:



> Some references strictly distinguish between parallelism and concurrency, but some do not clearly point out the difference. The concepts of parallelism and concurrency often appear in

> Unreal Engine's multi-threaded rendering architecture and API, so Unreal clearly distinguishes the meanings of the two.

- **Thread Pool**

The thread pool provides a new way of task concurrency. The caller only needs to pass in a set of parallel tasks and a grouping strategy, and then several threads in the thread pool can be used to execute tasks concurrently. The caller does not need to directly contact the calling and management details of the thread, which reduces the cost of the caller and improves the scheduling efficiency and throughput of the thread.

However, when creating a thread pool, several key design issues affect concurrency efficiency, such as: the number of threads available, efficient task allocation, and whether it is necessary to wait for a task to complete.

The thread pool can be customized or directly use the API provided by C++, operating system or third-party library.

2.1.3 Multithreading in C++

Before C++11, C++ had almost no support for multithreading, and only provided a few useless volatilekeywords. It was not until the C++11 standard that multithreading was truly incorporated into the C++ standard, and related keywords and the STL standard library were provided to enable users to implement cross-platform multithreading calls.

Of course, for users, multithreading can be implemented using C++11 thread library, or you can customize the thread library according to the multithreading API provided by the specific system platform, or use third-party libraries such as ACE, boost:: thread, etc. There are several advantages to using C++'s own multithreading library. First, it is easy to use and has few dependencies; second, it is cross-platform and does not require attention to the underlying system.

2.1.3.1 C++ Multithreading Keywords

- **thread_local**

thread_local is the key to implementing thread local storage in C++. A variable with this keyword means that each thread has its own copy of the data and will not share the same data, thus avoiding data competition.

C11 keywords_Thread_localare used to define thread-local variables. The header files<threads.h> definethread_localsynonyms for the above keywords. For example:

```
#include<threads.h>
thread_localintfoo =0;
```

Keywords introduced in C++11thread_localare used in the following situations:

1. Namespace (global) variables.

2. File static variables. 3. Function static variables. 4. Static member variables. In addition, different compilers provide their own methods for declaring thread-local variables:

```
// Visual C++, Intel C/C++ (Windows systems), C++Builder, Digital Mars C++ __declspec(thread)int
number;

// Solaris Studio C/C++, IBM XL C/C++, GNU C, Clang, Intel C++ Compiler (Linux systems) __threadintnumber;


//    C++ Builder
int    __thread number;
```

- **volatile**

A variable with the volatile modifier means that its value in memory may change at any time, and it also tells the compiler not to make any optimizations. Each time the value of this variable is used, it must be read from memory, and the value of the register should not be used directly.

Let's take a concrete example. Suppose there is the following code segment:

```
inta =10;
volatileint*p = &a; int
       b, c;
b=    * p;
c   =  * p;
```

Ifpthere is no memory oncep volatilemodification, theb = *psum , so the value of thebsumcmust be c = *ponly needs to be taken from 10.

If we considervolatilethe impact, assuming that after the statement is executed value of is b = *p,pthe modified by other threads, then the execution will read the value memory again. Atc t=h *isp from the time, the value is no longer 10, but a new value.p c

However, volatile cannot solve the synchronization problem of multiple threads and is only suitable for the following three situations:
1. Occasions related to signal handlers.

2. Situations related to memory mapped hardware.

3. Situations related to non-local jumps (setjmpand). longjmp

- **std::atomic**

Strictly speaking,atomicit is not a keyword, but a template class of STL that supports atomic operations of specified types.

Using an atomic type means that the read and write operations of instances of this type are atomic and cannot be cut by other threads, thus achieving the goals of thread safety and synchronization.

Some readers may be curious about why atomic operations are also required for basic types of operations. For example:

```
intcnt =0;
autof = [&]{cnt++;};
std::thread t1{f}, t2{f}, t3{f};
```

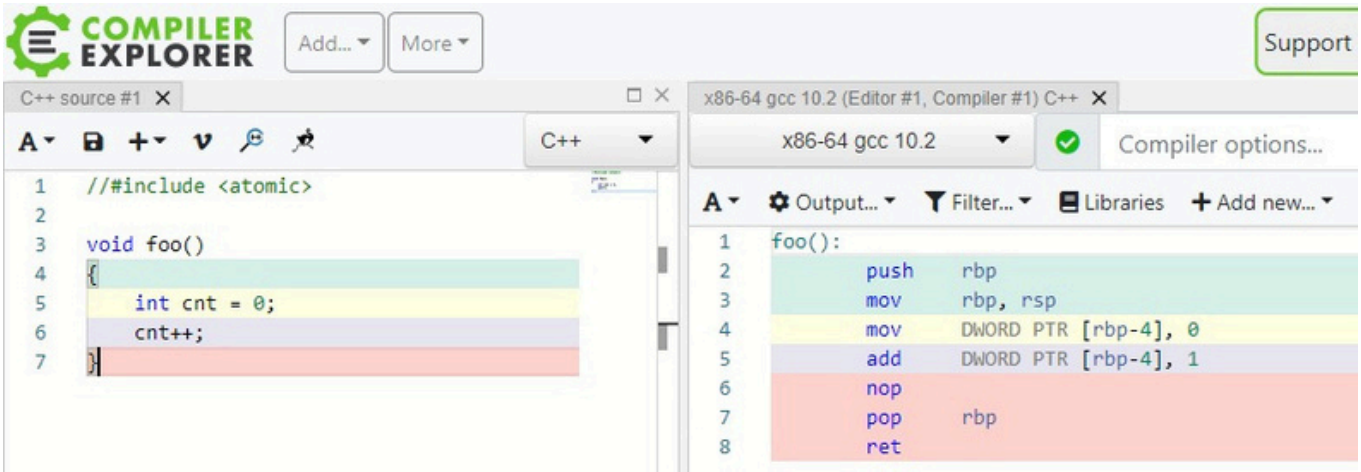
The above three threads call the function at the same timef.The function only executescnt++. In the C++ dimension, it seems that there is only one execution statement. In theory, there should be no synchronization problem. However, after being compiled into assembly instructions, there will be multiple instructions, which will cause thread context switching in multi-threading and cause unpredictable behavior.

To avoid this, you need to addatomicthe type:

```
std::atomic<int> cnt{0}; autof =        //GivecntAdd atomic operations.
[&]{cnt++;};
std::thread t1{f}, t2{f}, t3{f};
```

After addingatomic,the results of all threads after execution are certain, and the variables can be counted normally.atomicThe implementation mechanism is similar to the critical section, but it is more efficient than the critical section.

To further illustrate that a single C++ statement may generate multiple assembly instructions, you can useCompiler Explorer to view the instructions after C++ assembly in real time:



Compiler Explorer dynamically compiles the C++ statements on the left into assembly instructions. After the C++ code shown in the figure above is compiled, there may be one-to-many assembly instructions, which confirms the necessity of atomic operations.

By making full usestd::atomicof the features and interfaces, many non-blocking, lock-free, threadsafe data structures and algorithms can be implemented. For further reading on this point, I strongly recommend"C++ Concurrency In Action" .

2.1.3.2 C++ Threads

The C++ thread type isstd::thread,and the interface it provides is as follows:

interface	Analysis
join	Join the main thread, so that the main thread is forced to wait for the thread to finish executing.
detach	Detach from the main thread so that the main thread does not need to wait for the thread to finish executing.
swap	Exchanges a thread object with another thread.
joinable	Query whether the main thread can be joined.
get_id	Gets a unique identifier for this thread.
native_handle	Returns the implementation layer's thread handle.
hardware_concurrency	Static interface, returns the number of concurrent threads supported by the hardware.

Example of use:

```
#include    <iostream>
#include    <thread>
#include    <chrono>

void foo()
{
       // simulate expensive operation
       std::this_thread::sleep_for(std::chrono::seconds(1));
}

int main()
{
       std::cout<<"starting thread...\n";
       std::threadt(foo);//Construct a thread object and pass in the function to be executed.


       std::cout<<"waiting for thread to finish..."<<std::endl; t.join();//Join the main thread so
       that the main thread must wait for this thread to finish executing.

       std::cout<<"done!\n";
}
```

Output:

```
starting thread...
waitingforthread to finish...
```

```
done!
```

If you need to synchronize data between the calling thread and the new thread, you can use C++'s std::promiseandstd::futuremechanisms. Sample code:

```
#include    <vector>
#include    <thread>
#include    <future>
#include    <numeric>
#include    <iostream>

void accumulate(std::vector<int>::iterator                  first,
                std::vector<int>::iterator                  last,
                std::promise<int> accumulate_promise)
{
    intsum =std::accumulate(first, last,0);
    accumulate_promise.set_value(sum);// Notify future
}

int main()
{
    // Demonstrate using promise<int> to transmit a result between threads. std::vector<int>
    numbers = {1,2,3,4,5,6}; std::promise<int> accumulate_promise;

    std::future<int> accumulate_future = accumulate_promise.get_future(); std::thread
    work_thread(accumulate, numbers.begin(), numbers.end(),
                                        std::move(accumulate_promise));

    // future::get() will wait until the future has a valid result and retrieves it. // Calling wait() before get() is not
    needed
    //accumulate_future.wait(); // wait for result
    std::cout<<"result = "<< accumulate_future.get() <<'\n'; work_thread.join();// wait
    for thread completion
}
```

Output:

```
Result=twenty one
```

However,std::threadthe execution is not guaranteed to be asynchronous and may also be executed in the current thread.

If you need to force asynchrony, you can use it std::async.It can specify two asynchronous modes: std::launch::asyncandstd::launch::deferred. The former means using a new thread to execute the task asynchronously, and the latter means executing it in the current thread and it will be delayed. Usage example:

```
#include    <iostream>
#include    <vector>
#include    <algorithm>
#include    <numeric>
#include    <future>
#include    <string>
```

```
#include<mutex>

std::mutex      m;
structX{
      void foo(inti,conststd::string& str) {
            std::lock_guard<std::mutex>lk(m); std::cout<< str
            <<' '<< i <<'\n';
      }
      void bar(conststd::string& str) {
            std::lock_guard<std::mutex> std::       lk(m);
            cout<< str <<'\n';
      }
      int operator()(inti) {
            std::lock_guard<std::mutex> std::       lk(m);
            cout<< i <<'\n'; returni +10;

      }
};

template <typename RandomIt>
int parallel_sum(RandomIt beg, RandomIt end) {

      autolen = end - beg; if(len <
      1000)
            return std::accumulate(beg, end,0);

      RandomIt mid = beg + len/2;
      autohandle =std::async(std::launch::async,
                                          parallel_sum<RandomIt>,          mid,    end);
      intsum = parallel_sum(beg, mid); returnsum
      + handle.get();
}

int main()
{
      std::vector<int>v(10000,1);
      std::cout<<"The sum is "<< parallel_sum(v.begin(), v.end()) <<'\n';

      X x;
      // Calls (&x)->foo(42, "Hello") with default policy: // may print "Hello 42"
      concurrently or defer execution autoa1 =std::async(&X::foo, &x,42,"Hello");
      // Calls x.bar("world!") with deferred policy

      // prints "world!" when a2.get() or a2.wait() is called
      autoa2 =std::async(std::launch::deferred, &X::bar, x,"world!"); // Calls X()(43); with async
      policy // prints "43" concurrently

      autoa3 =std::async(std::launch::async, X(),43); a2.wait();
                                                // prints "world!"
      std::cout<< a3.get() <<'\n';// prints "53"
}// if a1 is not done at this point, destructor of a1 prints "Hello 42" here
```

Execution Result:

```
Thesum is10000 43

Hello42
```

```
world!
53
```

In addition, C++20 already supports lightweight coroutines. The relevant keywords are:co_await, co_return,co_yield.The concepts and usage are exactly the same as those of scripting languages such as C#, but the behavior and implementation mechanism may be slightly different, which will not be discussed in this article.

2.1.3.3 C++ multithreaded synchronization

There are many mechanisms for thread synchronization, and C++ supports the following:

- **std::atomic**

[2.1.3.1 C++ multithreading keywords](#2.1.3.1 C++ multithreading keywords) has been std::atomicanalyzed in detail to prevent data races between multiple threads. In addition, it also Provides a variety of interfaces and status queries to synchronize atomic data more finely and efficiently. Common interfaces and analysis are as follows:

Interface Name	Analysis
is_lock_free	Checks whether an atomic object is lock-free.
store	Stores a value into an atomic object.
load	Loads a value from an atomic object.
exchange	Get the value of an atomic object and replace it with the specified value.
compare_exchange_weak, compare_exchange_strong	Compare the value of the atomic object with the expected value (expected). If they are the same, replace it with the desired value and return ittrue;if they are different, load the value of the atomic object to the expected value (expected) and return itfalse.The weak mode will not block the calling thread, while the strong mode will block the calling thread until the value of the atomic object is the same as the expected value (expected).
fetch_add, fetch_sub, fetch_and, fetch_or, fetch_xor	Get the value of atomic objects and perform operations such as addition and subtraction on them.
operator ++, operator --,	Atomic objects respond to various operators, and the meaning of the operators is the same as that of ordinary variables.

Interface Name	Analysis
operator +=, operator -=, ...	

In addition, C++20 also supports synchronization interfaces such as wait, notify_one, and notify_all.

The interface can be usedcompare_exchange_weakto easily implement a thread-safe, non-blocking data structure. Example:

```
#include    <atomic>
#include    <future>
#include    <iostream>

template<typename     T>
struct    node
{
      T data;
      node* next;
      node(constT& data) : data(data), next(nullptr) {}
};

template<typename T>
classstack
{
  public:
      std::atomic<node<T>*> head; public:       //   Stack Head,    Use atomic operations.

      //Push Operation
      void push(constT& data) {

            node<T>* new_node = new node<T>(data);

            //Use the original head pointer as the next node of the new node.
            new_node->next = head.load(std::memory_order_relaxed);

            //Compare the new node with the old head node. Ifnew_node->next==head,Indicates that other threads have not modifiedhead, You canhead Replace withnew_node,Thus completingpushoperate.
            //On the contrary, ifnew_node->next!=head,Indicates that other threads have modifiedhead,Modify other threadsheadSave to new_node->next,Continue the detection cycle.
            while(!head.compare_exchange_weak(new_node->next, new_node,
                                              std::memory_order_release, std
                                              ::memory_order_relaxed))
                  ;//Empty loop body
      }
};

int main()
{
      stack<int> s;

      autor1 =std::async(std::launch::async, &stack<int>::push, &s,1); autor2 =std::async(std
      ::launch::async, &stack<int>::push, &s,2); autor3 =std::async(std::launch::async, &stack<int
      >::push, &s,3);
```

```
    r1.wait();
    r2.wait();
    r3.wait();

    // print the stack's values
    node<int>* node = s.head.load(std::memory_order_relaxed); while(node)

    {
        std::cout<< node->data <<" "; node =
        node->next;
    }
}
```

Output:

```
23 1
```

It can be seen that atoms and their interfaces can be used to easily synchronize multiple threads. Moreover, since multiple threads are pushed into the stack asynchronously, the elements of the stack are not necessarily consistent with the order of encoding.

The above code also involves marking the memory access order:

- Sorting sequence is consistent.
- Acquire-release sequence (memory_order_consume, memory_order_acquire, memory_order_release, and memory_order_acq_rel).
- Free sequence (memory_order_relaxed).

For more information on this, seethe memory barriers in the first article or section 5.3 Synchronization and forced ordering in C++ concurrency in action.

- **std::mutex**

std::mutex is a mutex. It will enter the critical section within its scope, so that the code snippet can only be accessed by one thread at a time. When other threads try to execute the snippet, they will be blocked. std::mutex is often used withstd::lock_guardExample code:

```
#include    <iostream>
#include    <map>
#include    <string>
#include    <chrono>
#include    <thread>
#include    <mutex>

std::map<std::string,std::string> g_pages; std::mutex
g_pages_mutex;                          //Declare a mutex

void save_page(conststd::string&url) {

    // simulate a long page fetch
    std::this_thread::sleep_for(std::chrono::seconds(2));
```

```
    std::stringresult ="fake content";

    //Cooperatestd::lock_guardUse, can enter and release the mutex
    in time. std::lock_guard<std::mutex>guard(g_pages_mutex);
    g_pages[url] = result;
}

int main()
{
    std::thread    t1(save_page,    "http://foo");
    std::thread    t2(save_page,    "http://bar");
    t1.join();
    t2.join();

    // safe to access g_pages without lock now, as the threads are joined for(constauto&pair:
    g_pages) {
        std::cout<<pair.first <<" => "<<pair.second <<'\n';
    }
}
```

Output:

```
http://bar => fake content http://
foo => fake content
```

In addition, manual std::mutexlocking and unlocking can implement some special behaviors, such as waiting for a certain mark:

```
#include    <chrono>
#include    <thread>
#include    <mutex>

boolflag;
std::mutex    m;

void wait_for_flag()
{
    std::unique_lock<std::mutex>lk(m);//Here we usestd::unique_lockRather thanstd::lock_guard. std::unique_lockIt is possible to try to acquire the lock. If it is currently locked by other threads, it will be delayed until the other threads release it before acquiring the lock.
    while(!flag)
    {
        lk.unlock();//Unlocking the mutex
        std::this_thread::sleep_for(std::chrono::milliseconds(100));//Hibernation100msDuring this time, other threads can enter the mutex to changeflagmark.
        lk.lock();          //Relock the mutex
    }
}
```

- **std::condition_variable**

std::condition_variableBothstd::condition_variable_anyare condition variables, both are implemented in the C++ standard library, and both need to be used with mutexes. Since they

std::condition_variable_anyare more general, they will incur more overhead in performance. Therefore, they should be considered firststd::condition_variable.

By using the condition variable interface and the mutex, you can easily perform operations such as waiting and notification between threads. Example:

```
#include    <iostream>
#include    <string>
#include    <thread>
#include    <mutex>
#include    <condition_variable>

std::mutex m;
std::condition_variable std::       cv;         //Declaring a condition variable
stringdata; boolready =false;
boolprocessed =false;


void worker_thread()
{
        //Wait until the main thread changesreadyfor
        true. std::unique_lock<std::mutex>lk(m);
        cv.wait(lk, []{returnready;});

        //Obtained the lock on the mutex
        std::cout<<"Worker thread is processing data\n"; data +="after
        processing";

        //Send data to the main thread
        processed =true;
        std::cout<<"Worker thread signals data processing completed\n";

        //Manually unlock so that the main thread can acquire the lock.
        lk.unlock();
        cv2.notify_one();
}

int main()
{
        std::threadworker(worker_thread);

        data ="Example data";
        // send data to the worker thread {

                std::lock_guard<std::mutex>lk(m); ready =true;

                std::cout<<"main() signals data ready for processing\n";
        }
        cv2.notify_one();

        // wait for the worker {

                std::unique_lock<std::mutex> cv.wait(lk,l k[]({m);
                returnprocessed;});
        }
        std::cout<<"Back in main(), data = "<< data <<'\n';
```

```
    worker.join();
}
```

Output:

```
main() signalsdataready for processing Workerthread
is processingdata
Workerthread signalsdataprocessing completed Backinmain(),data=
Exampledataafter processing
```

- **std::future**

C++ future is a mechanism for accessing future return values, often used for multi-threaded synchronization. The types that can create futures are: std::async, std::packaged_task, std::promise. The future object can execute wait, wait_for, and wait_until to implement event waiting and synchronization. The sample code is as follows:

```
#include    <iostream>
#include    <future>
#include    <thread>

int main()
{
    //frompackaged_taskObtainedfuture
    std::packaged_task<int()>task([]{return7; });// wrap the function std::future<int> f1 =
    task.get_future();// get a future std::threadt(std::move(task));// launch on a thread


    //fromasync()Obtainedfuture
    std::future<int> f2 =std::async(std::launch::async, []{return8; });

    //frompromiseObtainedfuture
    std::promise<int> p;
    std::future<int> f3 = p.get_future();
    std::thread( [&p]{ p.set_value_at_thread_exit(9detach();

    //Waiting for allfuture
    std::cout<<"Waiting..."<<std::flush; f1.wait();

    f2.wait(); f3.wait(); std::cout<<"Done!\nResults are: "<< f1.get() <<' '<< f2.get()
    <<' '<< f3.get() '\n';

<<
    t.join();
}
```

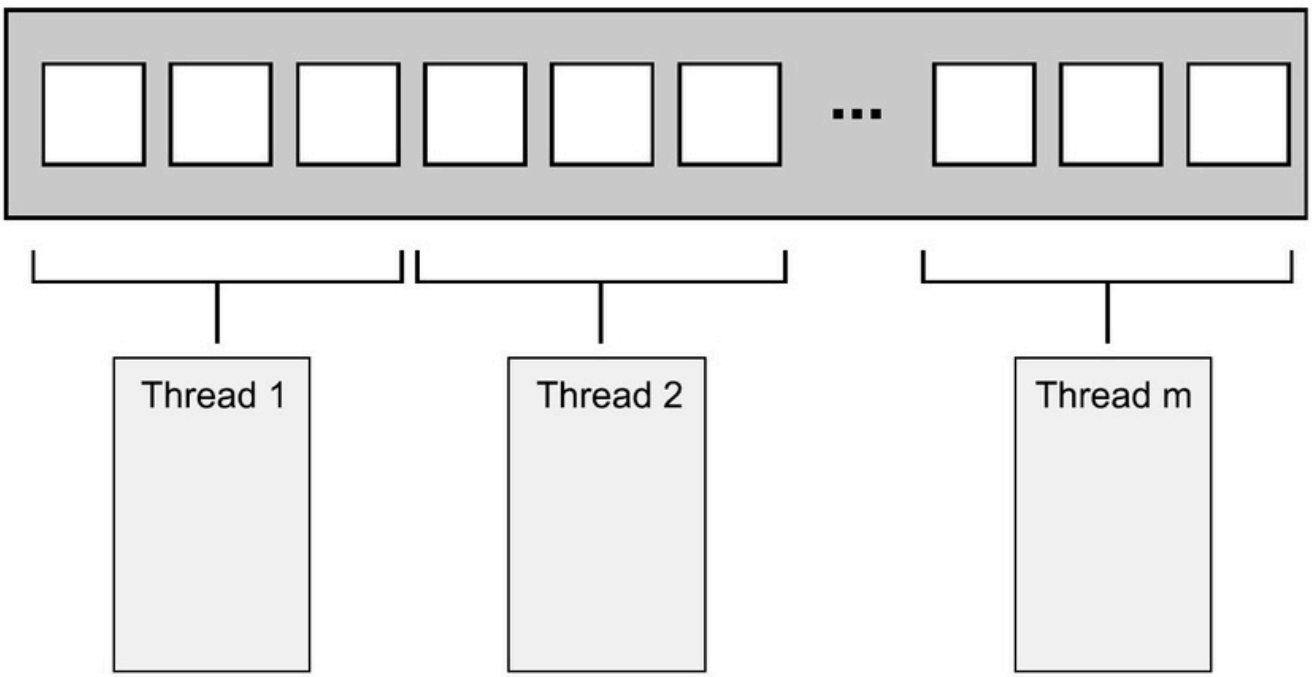
Output:

```
Waiting...Done!
Resultsare:7 8 9
```

2.1.4 Multithreading Implementation Mechanism

Multithreading can be divided into data parallelism and task parallelism according to the parallel content. Data parallelism means that different threads carry different data to execute the same logic. The most classic data parallel applications are MMX instructions, SIMD technology, Compute shaders, etc. Task parallelism means that different threads execute different logics, and the data can be the same or different. For example, game engines often put file loading, audio processing, network reception and even physical simulation into separate threads so that they can execute
different tasks in parallel.

If multithreading is divided according to the granularity and method, there are linear division, recursive division, task type division, etc.
The simplest application of the linear partitioning method is to divide the elements of a continuous array into several equal parts, and dispatch each piece of data to a thread for execution, such as parallelizationstd::for_eachand UEParallelFor.



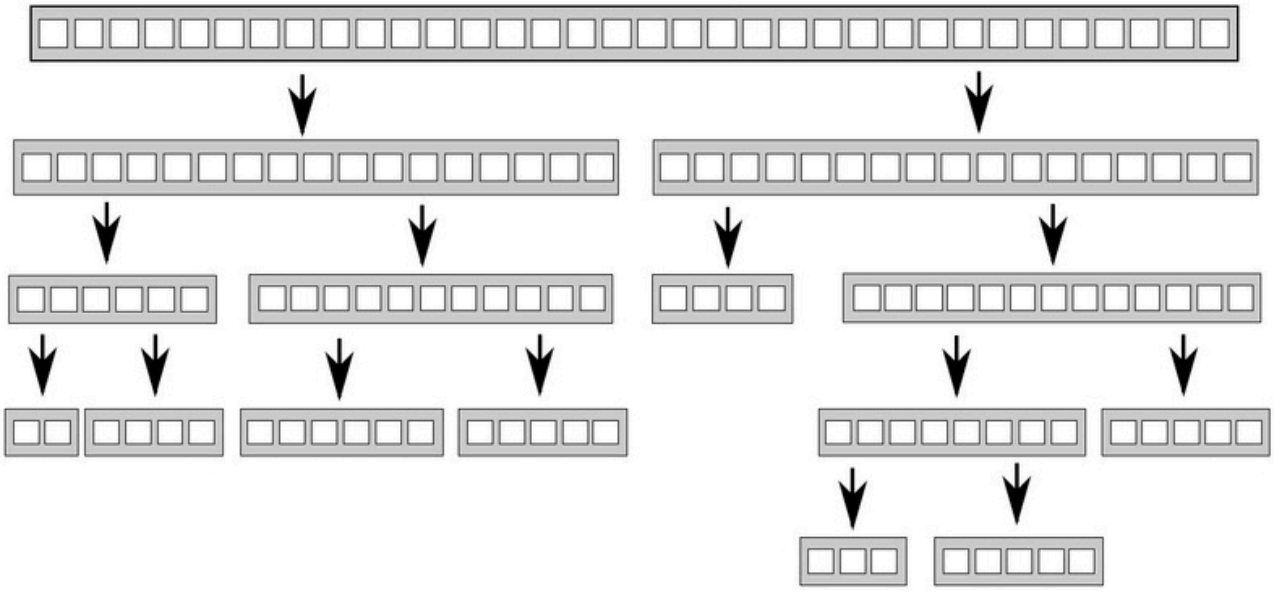
Schematic diagram of linear partitioning. Continuous data is divided into several equal parts, and then dispatched to several threads for parallel execution.

After the parallel execution of the linear partition is completed, the calling thread usually needs to merge and synchronize the parallel results.

Recursive partitioning is to divide continuous data into several parts according to a certain rule, and each part can be further divided into finer granularity until a certain rule stops the division. It is often used for quick sorting.

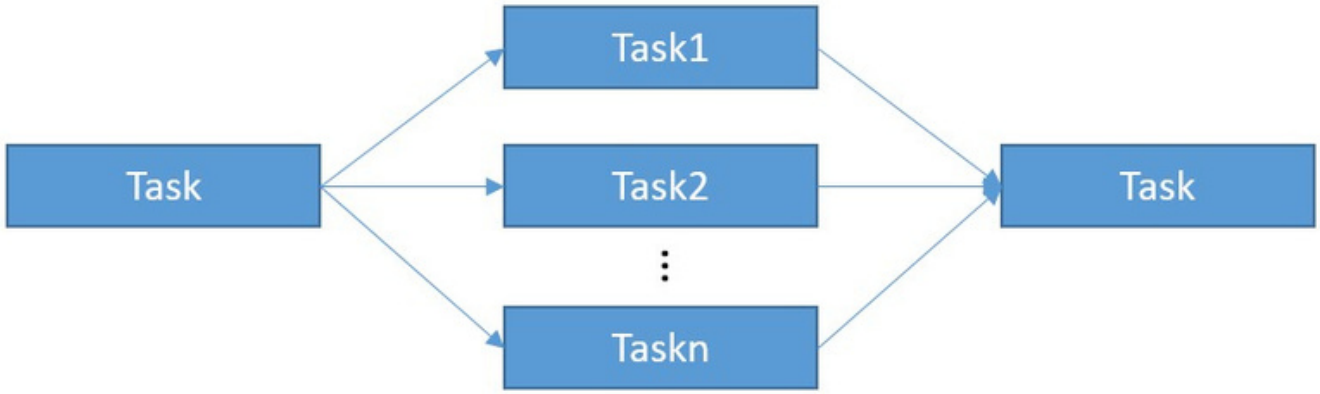
There are two basic steps in quick sort: divide the data into two halves before and after the pivot element, and then quickly sort the two halves before and after the pivot element again. Since you

can only know which items are before and after the pivot element after a sort, you cannot achieve parallelism by simply (linearly) dividing the data. When you want to parallelize this algorithm, it is natural to think of using recursion. Each level of recursion will call the quick_sort function multiple times because you need to know which elements are before and after the pivot element.



Schematic diagram of the recursive partitioning method.

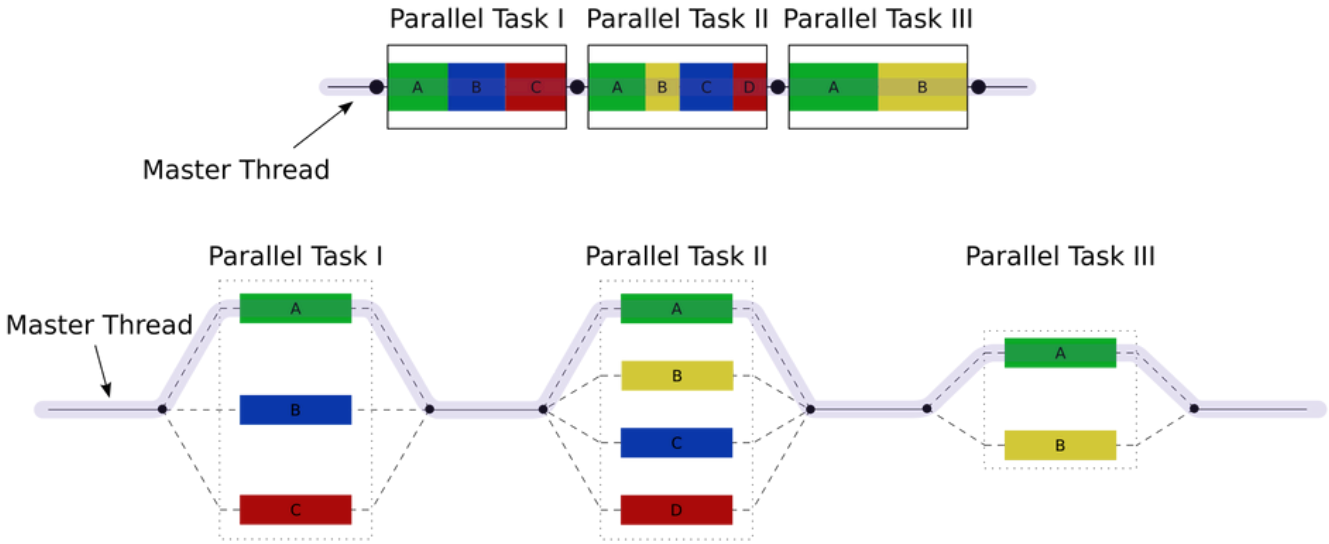
Divide the logic within a large framework into several subtasks, which are usually independent of each other but may have certain dependencies. Each task is dispatched to a thread for execution, which means true thread independence. Each thread only needs to focus on what it needs to do.



Schematic diagram of task division.

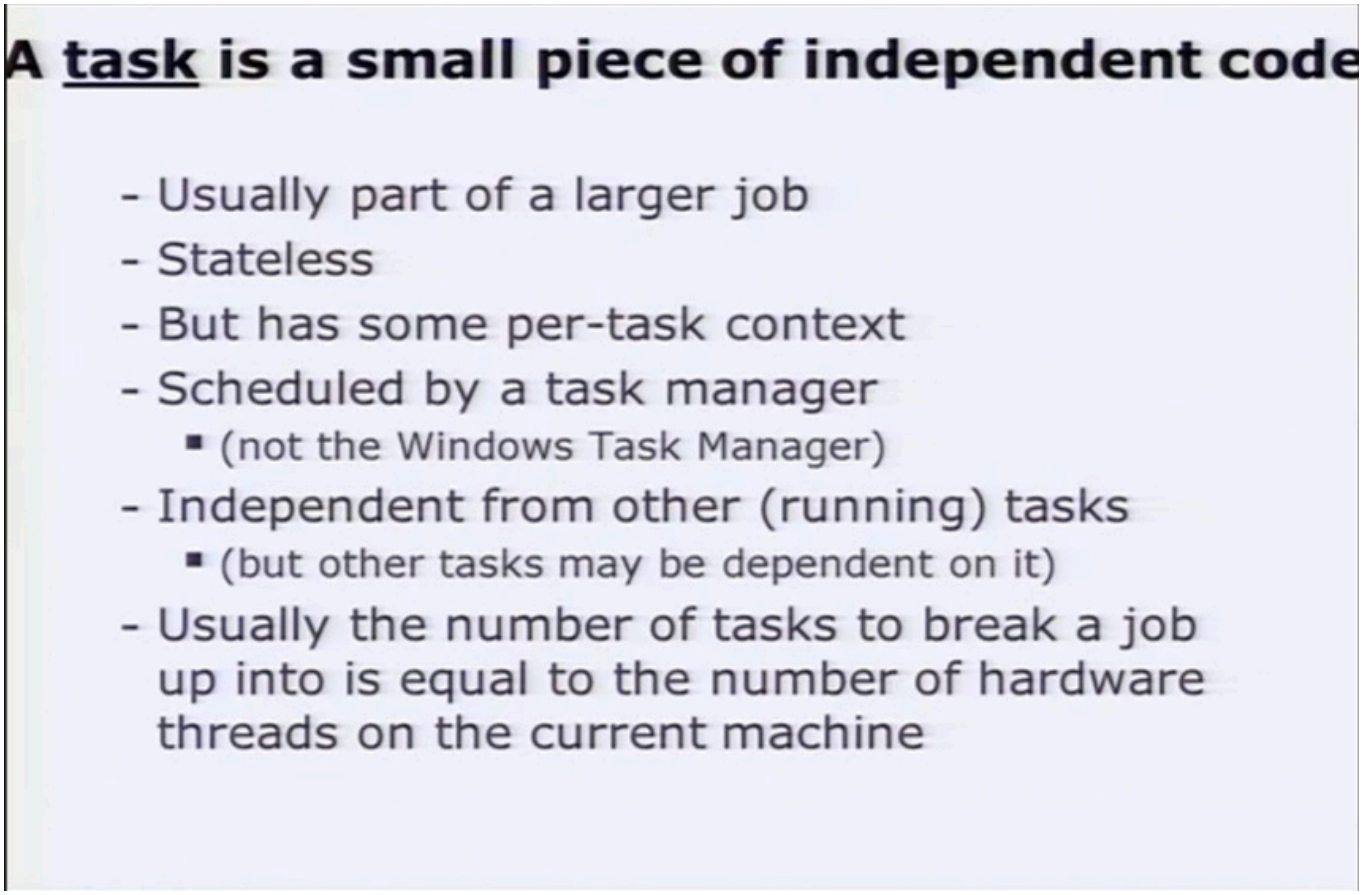
Reasonable arrangement and division of subtasks, reducing the dependencies and synchronization between them, is a powerful weapon to improve operation efficiency. However, to achieve this, it often requires careful design implementation and repeated debugging and modification.

The above implementation mechanism is often called**the Fork-Join parallel**model. Its operation mechanism is compared with the serial model as shown in the following figure:

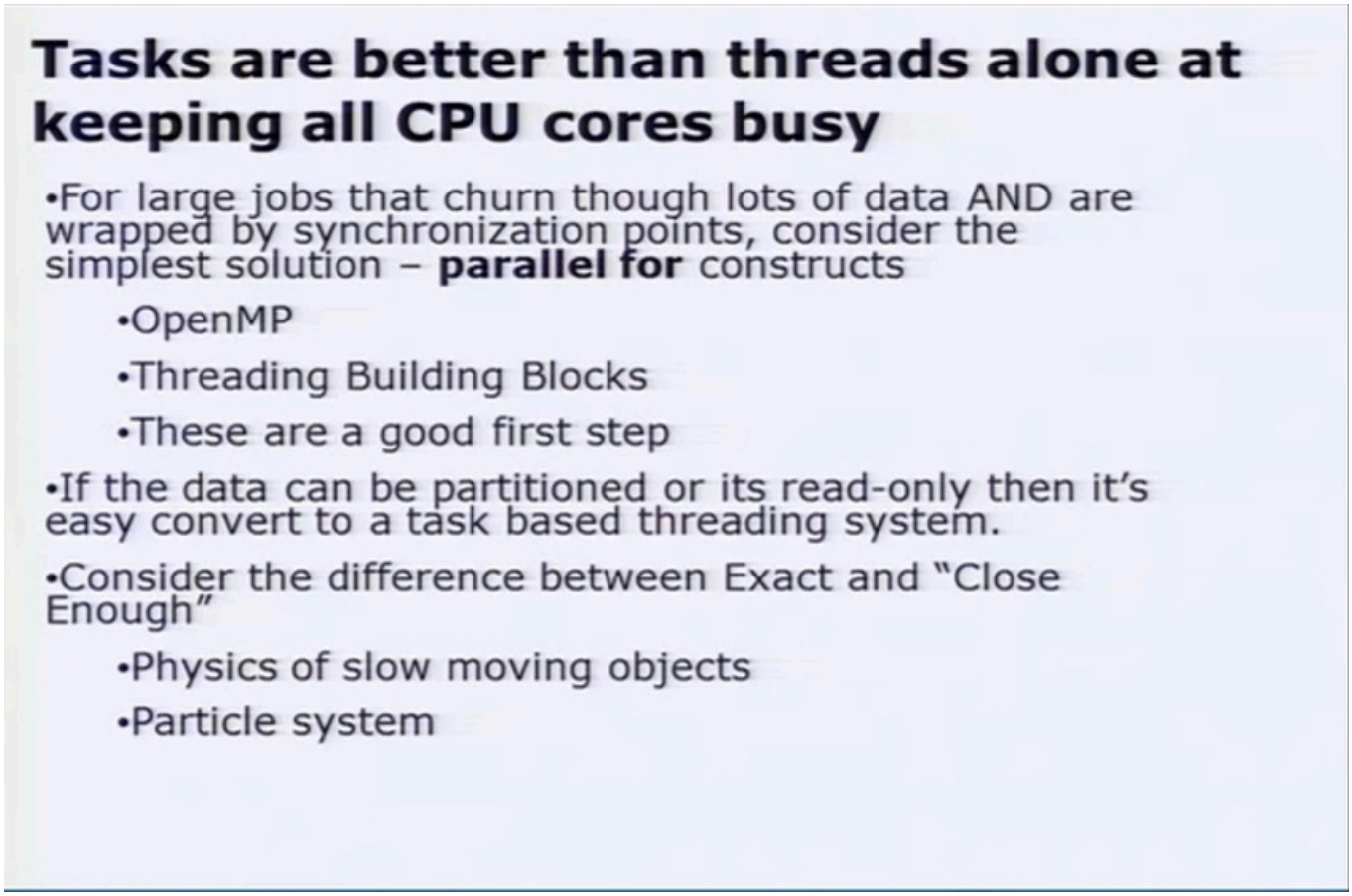


Top: Serial operation model; Bottom: Fork-Join parallel operation model.

The GDC2010 speechTask-based Multithreading - How to Program for 100 cores elaborates on how to use the Task-based multithreading operation mechanism:



Task-based multithreading is much better than thread-based architecture. It can make fuller use of multi-core advantages and keep each core busy:



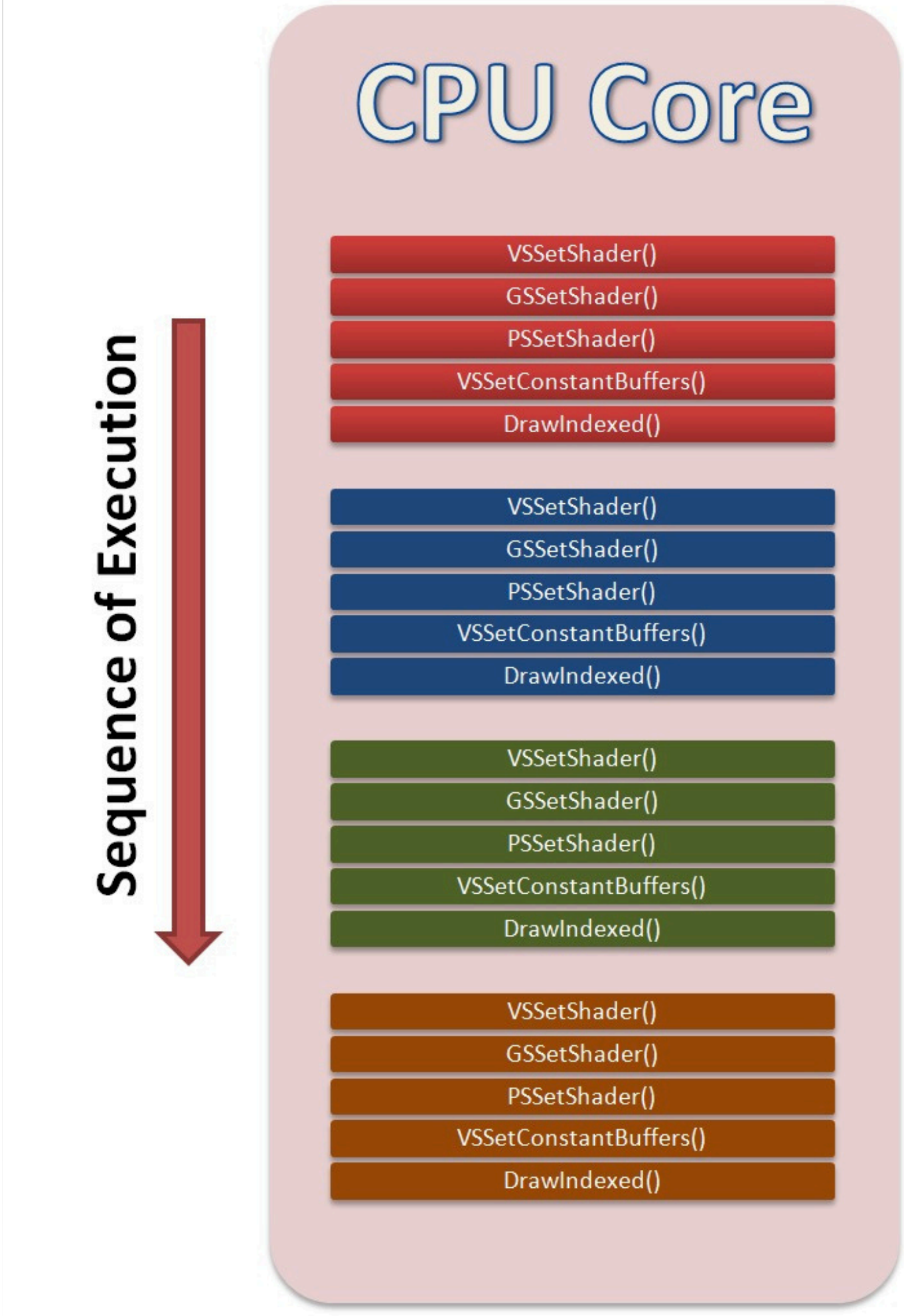
The article also mentions how to apply task-based multithreading to practical cases such as sorting and maze pathfinding.

2.2 Multithreading Features of Modern Graphics APIs

2.2.1 Multithreading Features of Traditional Graphics APIs

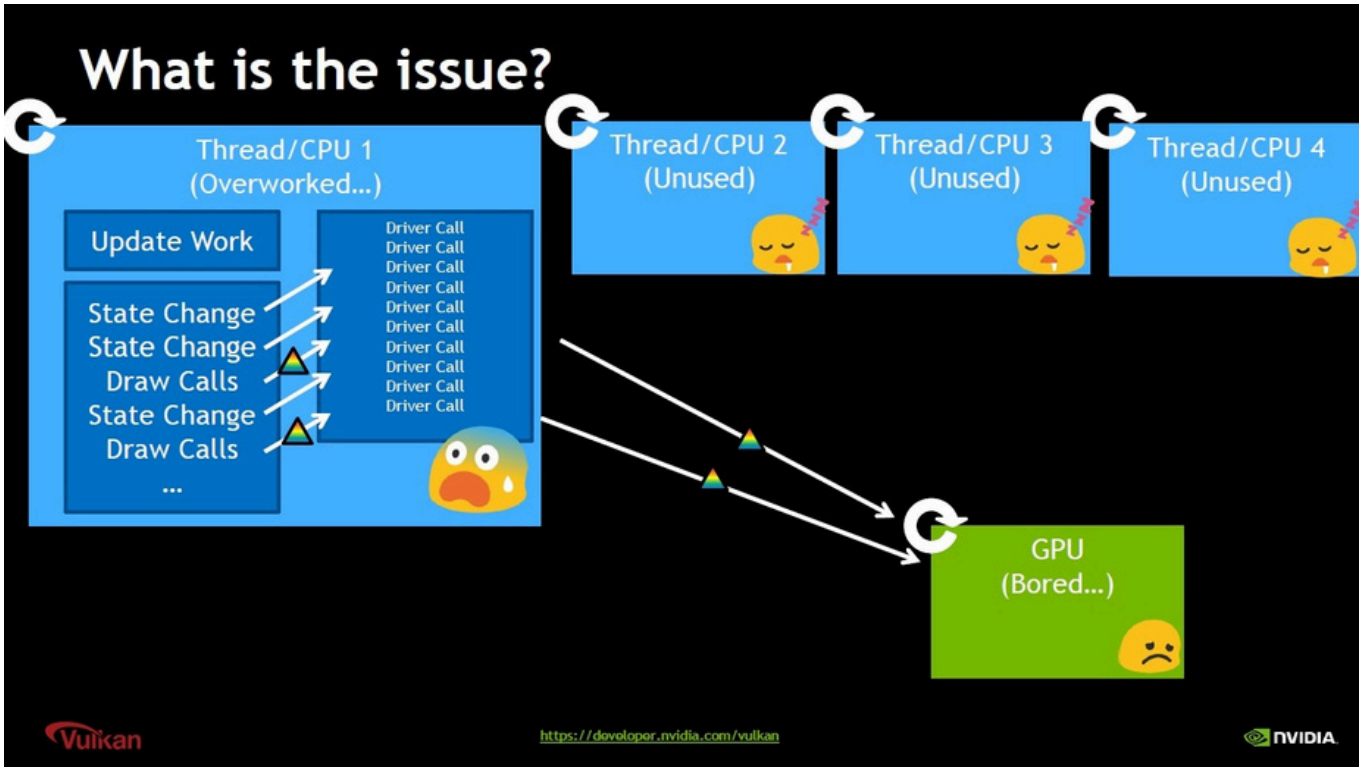
All drawing instructions in OpenGL and graphics API versions prior to DirectX10 are linear and blocking, which means that each call to the Draw interface will not return immediately and will block the calling thread. This CPU and GPU interaction mechanism has little impact on performance in the single-core era, but with the advent of the multi-core era, this interaction mechanism will obviously seriously affect performance.

If the game engine's renderer is still single-threaded, this often leads to CPU performance bottlenecks, hindering the use of multi-core computing resources to improve performance or enrich visualization content.


CPU Core
Sequence of Execution
VSSetShader()
GSSetShader()
PSSetShader()
VSSetConstantBuffers()
DrawIndexed()
VSSetShader()
GSSetShader()
PSSetShader()
VSSetConstantBuffers()
DrawIndexed()
VSSetShader()
GSSetShader()
PSSetShader()
VSSetConstantBuffers()
DrawIndexed()
VSSetShader()
GSSetShader()
PSSetShader()
VSSetConstantBuffers()
DrawIndexed()

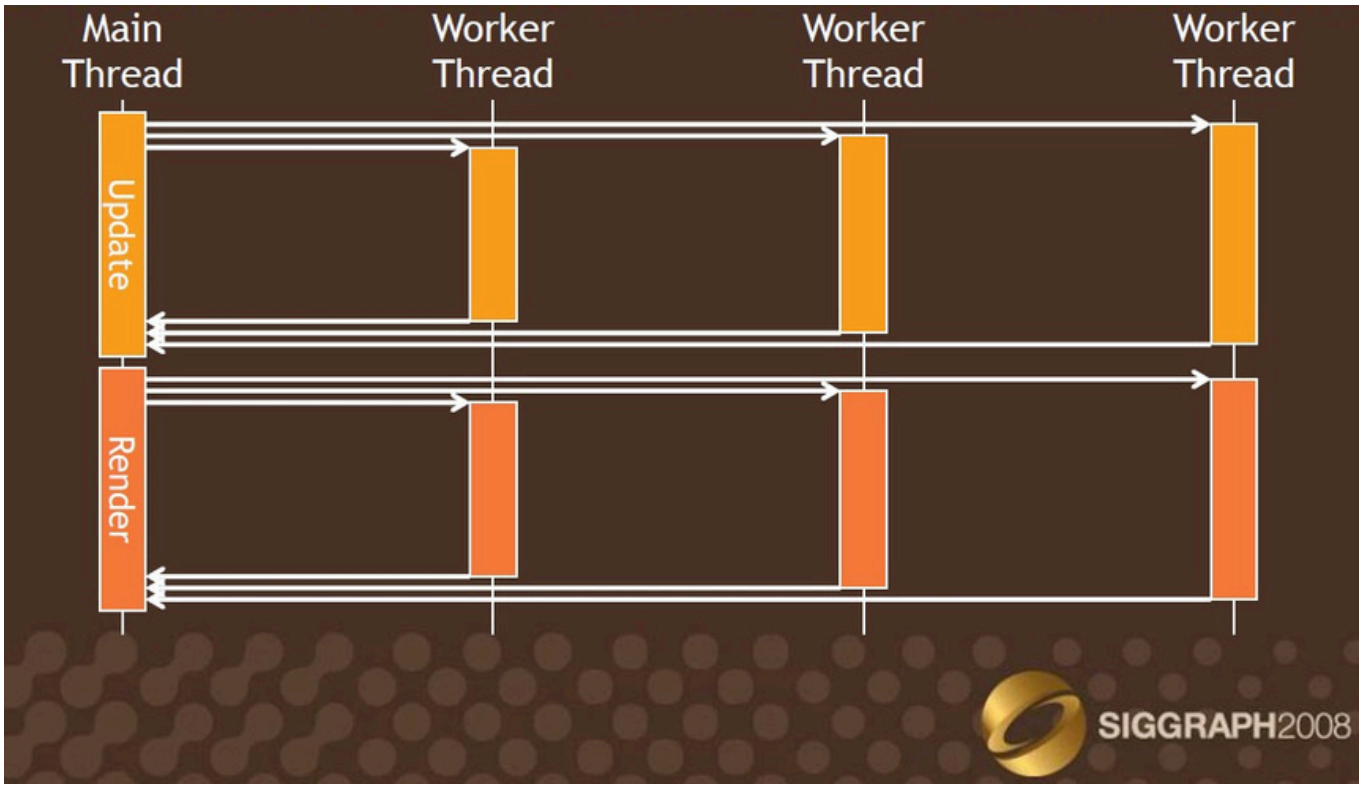
Schematic diagram of traditional graphics API linear execution of drawing instructions.

Single-threaded renderers often result in a single CPU core being fully utilized while other cores remain relatively idle and performance drops below a playable frame rate.



Traditional graphics APIs set the rendering state and call drawing instructions in a single-threaded single context, and the drawing instructions are blocking. The CPU and GPU cannot run in parallel, and other CPU cores will be in an idle waiting state.

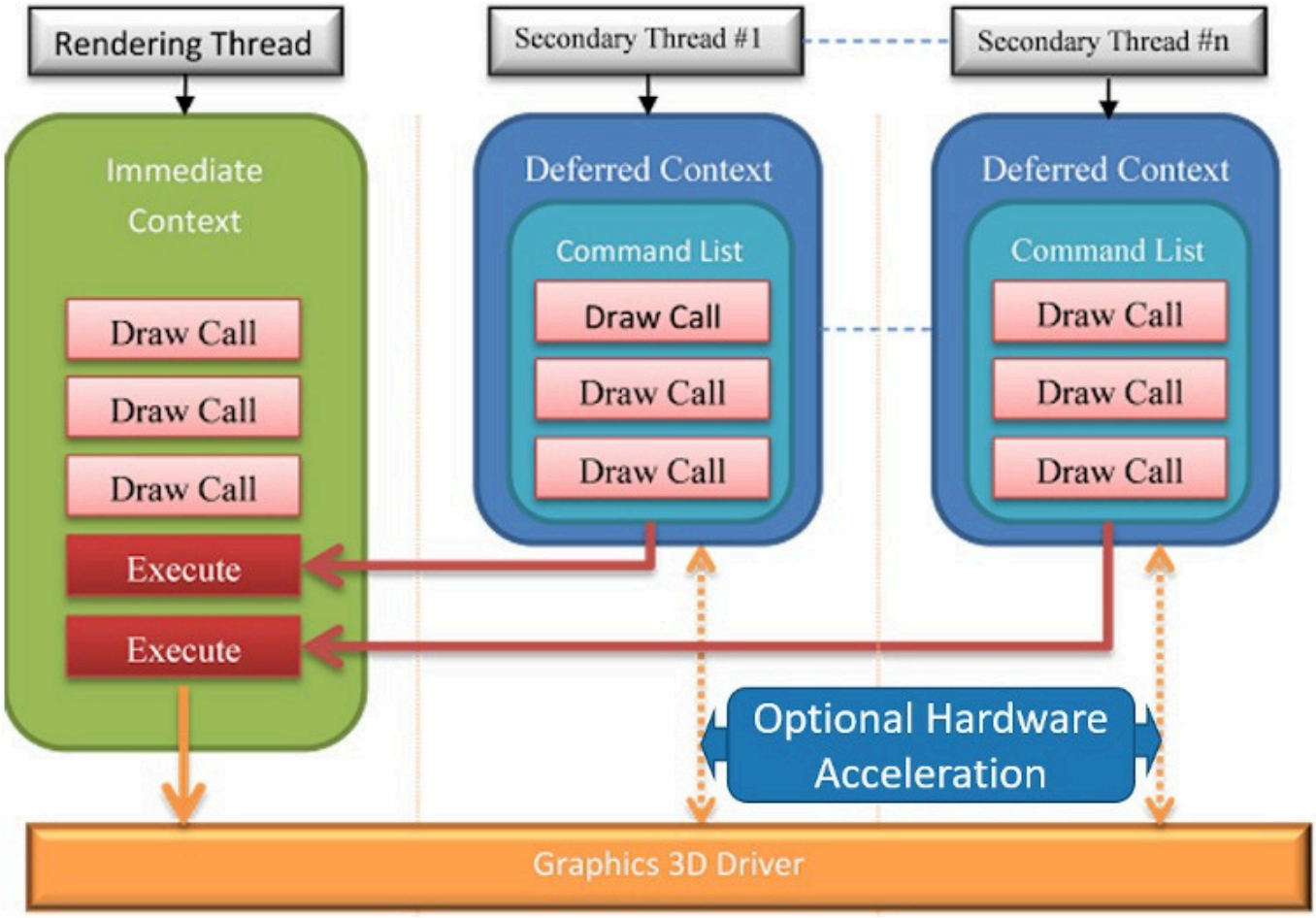
In these traditional graphics API architectures, multi-threaded rendering must be done from the software level, opening up multiple threads to process logic and rendering instructions separately, so as to relieve the mutual waiting coupling between the CPU and GPU. As early as SIGGraph2008, there was a talk (Practical Parallel Rendering with DirectX 9 and 10) that specifically explained how to implement software-level multi-threaded rendering in DirectX9 and 10. The core part is to record (Playback) D3D drawing commands (Command) at the software level.



A software-level multi-threaded rendering architecture proposed in Practical Parallel Rendering with DirectX 9 and 10.

However, this software-level command recording has many problems. It does not support some graphics APIs (such as status query), requires an additional command buffer to record drawing instructions, and cannot create real GPU resources in the command stage.

DirectX11 attempts to solve the problem of multi-threaded rendering from the hardware level. It supports two types of device contexts:**Immediate Context**and**Deferred Context**.Different deferred contexts can be used in different threads at the same time to generate a list of commands to be executed in the "immediate context". This multi-threaded strategy allows complex scenes to be decomposed into concurrent tasks.



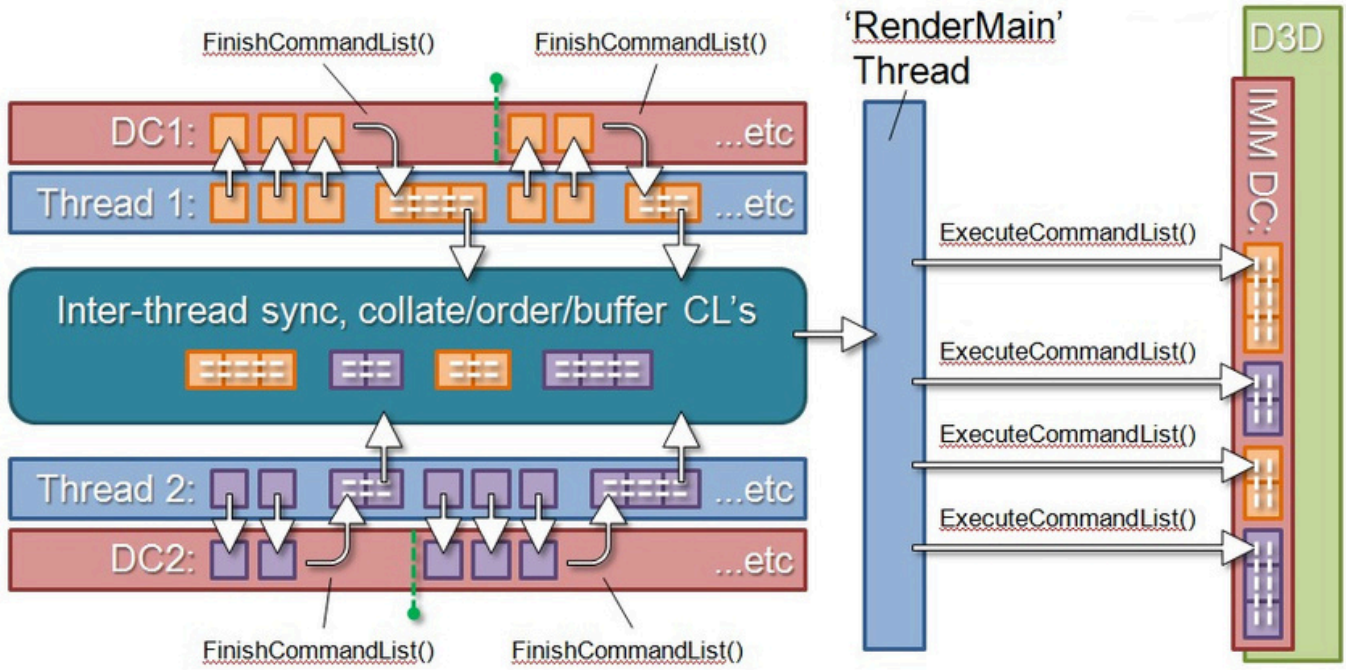
DirectX11's multithreading model.

Different deferred contexts can be used in different threads at the same time to generate command lists that will be executed in the immediate context. This multithreading strategy allows complex scenarios to be broken down into concurrent tasks. In addition, with the support of some drivers, deferred contexts can achieve hardware-level acceleration without having to execute the command list in the immediate context.

> **Why is it more efficient to use Deferred Context's Command List to record drawing commands in advance than to directly use Immediate Context to call drawing commands?** The answer is that the recorded commands are highly optimized within the Command List. Executing these optimized commands will significantly improve efficiency, which is much more efficient than directly calling the graphics API individually.

In D3D11, the commands in the command list are recorded quickly instead of being executed immediately. They are not actually executed by the GPU until the program calls the ExecuteCommandList method (call and return, no waiting). At this time, the CPU threads that use the deferred rendering device interface and the main rendering thread can do other things, such as continuing the collision detection, physical transformation, animation interpolation, lighting preparation, etc. of the next frame, so as to prepare for recording a new command list.
However, based on the multi-threaded architecture of DirectX11, since hardware-level acceleration is not necessarily supported, all instructions recorded in the Deferred Context together with the

instructions in the Immediate Context must be submitted to the GPU for execution by the same thread (usually the rendering thread).

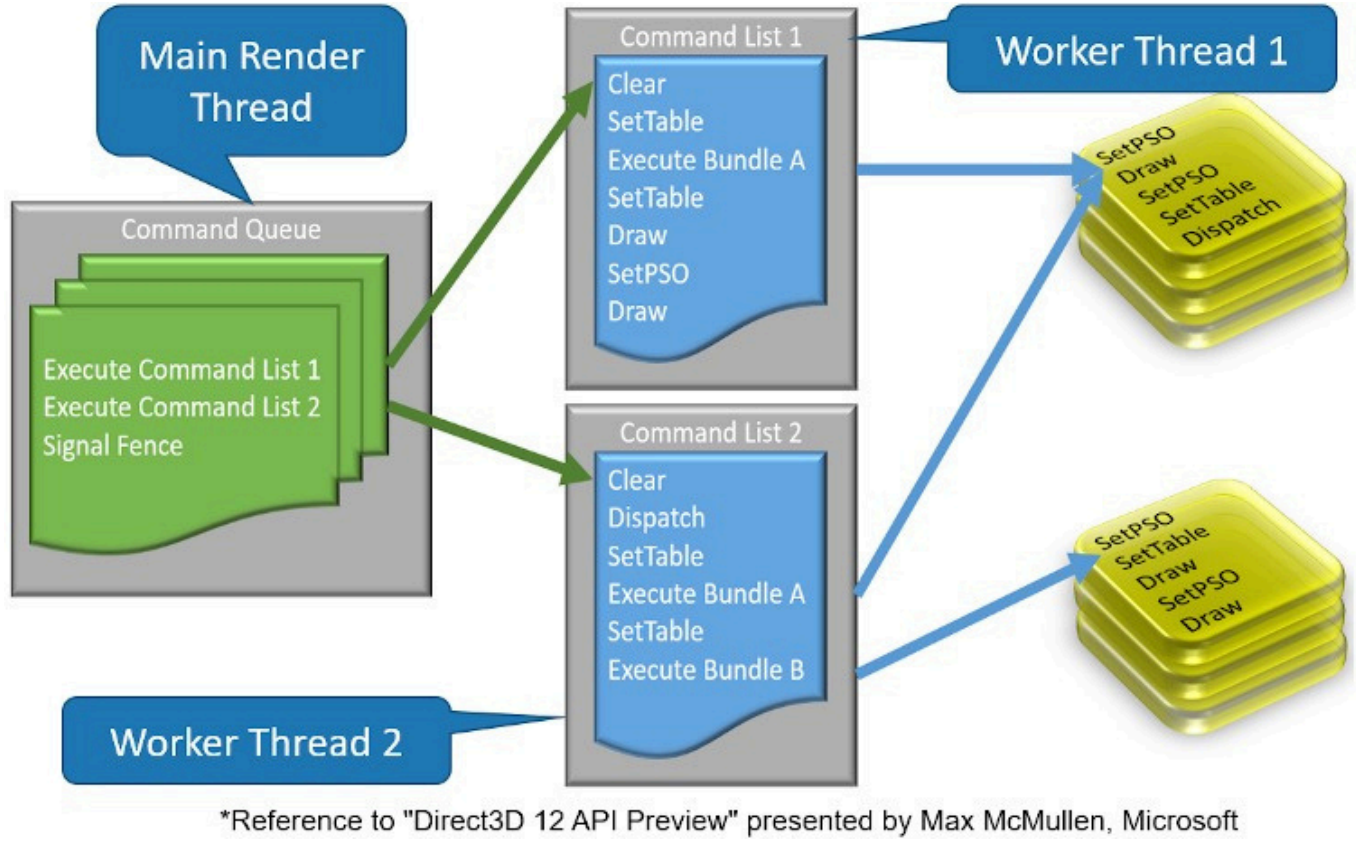


Schematic diagram of multi-threaded architecture under DirectX11.

This non-hardware supported multi-threaded rendering only saves some CPU time (multi-threaded recording instructions and drawing synchronous waiting), and cannot truly bring out the power of multi-threading from the hardware level.

2.2.2 Multithreading Features of DirectX12

Compared to the transitional pseudo-multithreading model of DirectX11 (called pseudo because most drivers at the time did not support DirectX11's hardware-level multithreading), DirectX 12 multithreading has been greatly improved by significantly reducing the additional overhead of API calls. It cancels the concept of device context in DirectX 11, directly uses Command List to call D3D APIs, and then submits the command list to the GPU through the command queue. In addition, all DirectX 12 graphics cards support hardware acceleration of DirectX 12 multithreading.



DirectX12's multithreading model.

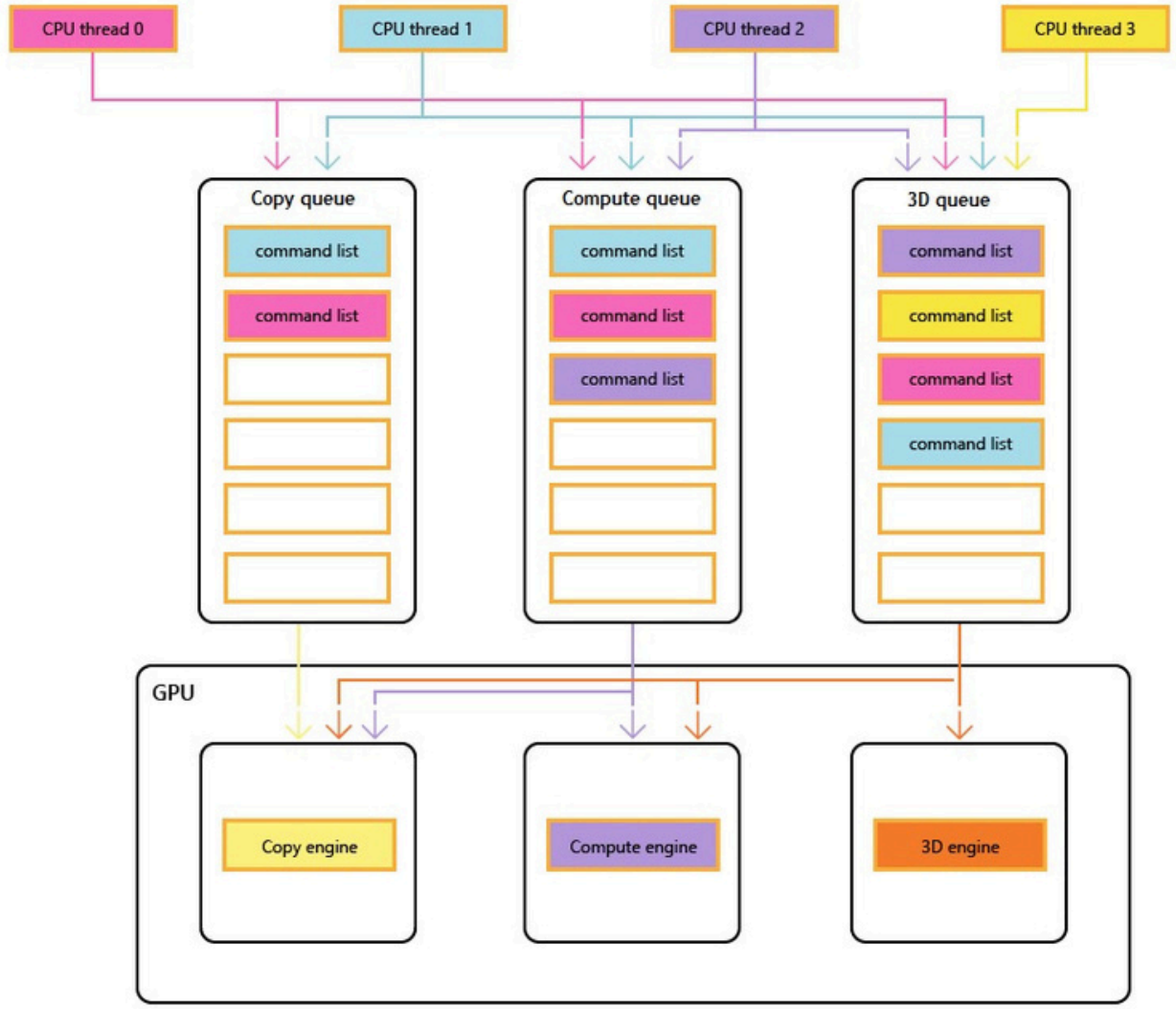
In principle, DirectX12 and DirectX11 multi-threaded rendering frameworks are similar, both of which complete multi-threaded rendering by recording command lists in different CPU threads and then executing them in a unified manner. They both fundamentally block the outrageous Draw Call synchronization call and instead use a completely asynchronous (parallel) execution method between the CPU and GPU, thereby greatly improving the overall rendering efficiency and performance.

For DirectX12, there are three types of command queues at the user level:**Copy Queue**,**Compute Queue**,and**3D Queue**.They can be executed in parallel and wait and synchronize through fences, signals, or barriers.

There are three types of engines at the GPU hardware level:**Copy Engine**,**Compute Engine,**and**3D Engine**. They can also execute in parallel and wait and synchronize through fences, signals, or barriers.

The command queue can drive several engines of the GPU hardware, but there are certain limitations. More specifically, the 3D Queue can drive three engines of the GPU hardware, the Compute Queue can only drive the Compute Engine and the Copy Engine, and the Copy Queue can only drive the Copy Engine.

At the CPU level, there can be several threads, each thread can create several command lists (Command List), and each command list can enter one of the three command queues. When these commands are executed by the GPU, the commands in each command list will be pushed into different GPU engines so that they can be executed in parallel. (Figure below)

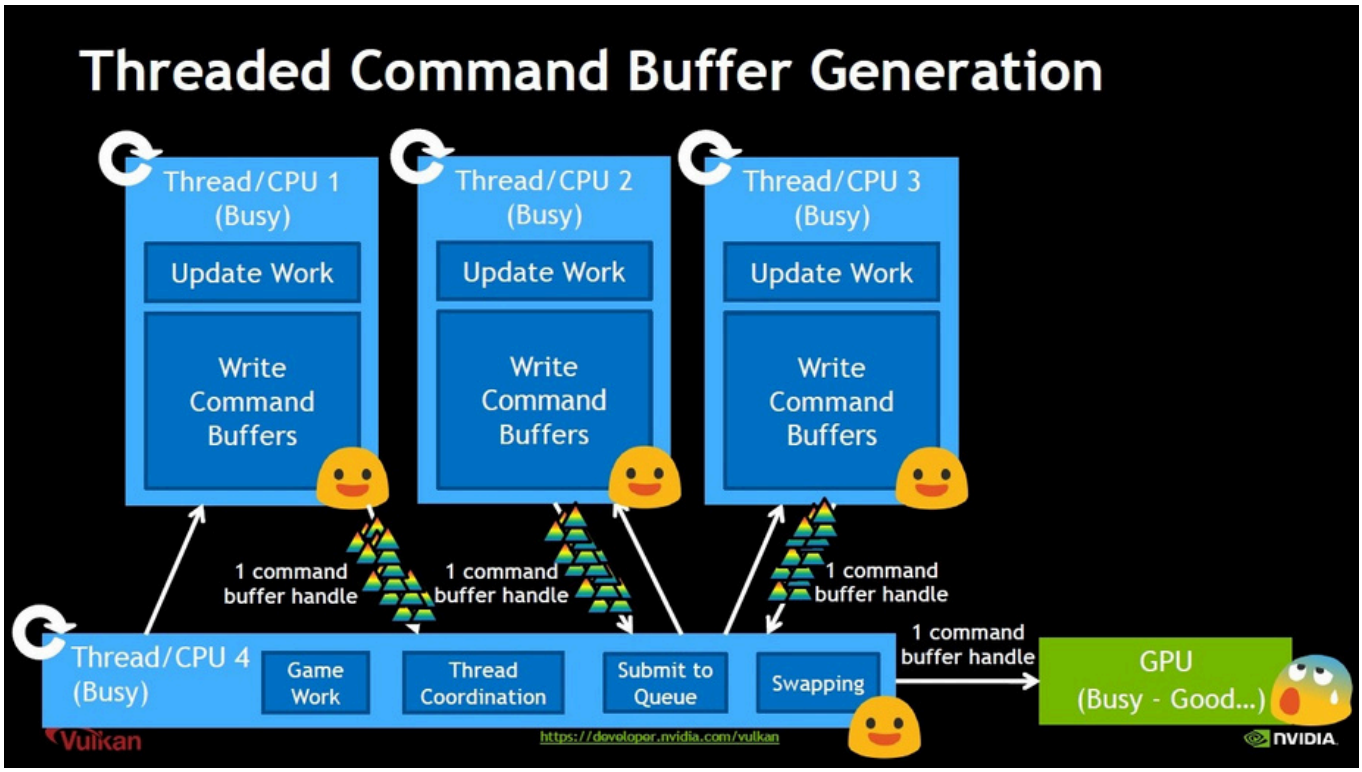


Schematic diagram of the operating mechanism between CPU threads, command lists, command queues, and GPU engines in DirectX12.

2.2.3 Vulkan's Multithreading Features

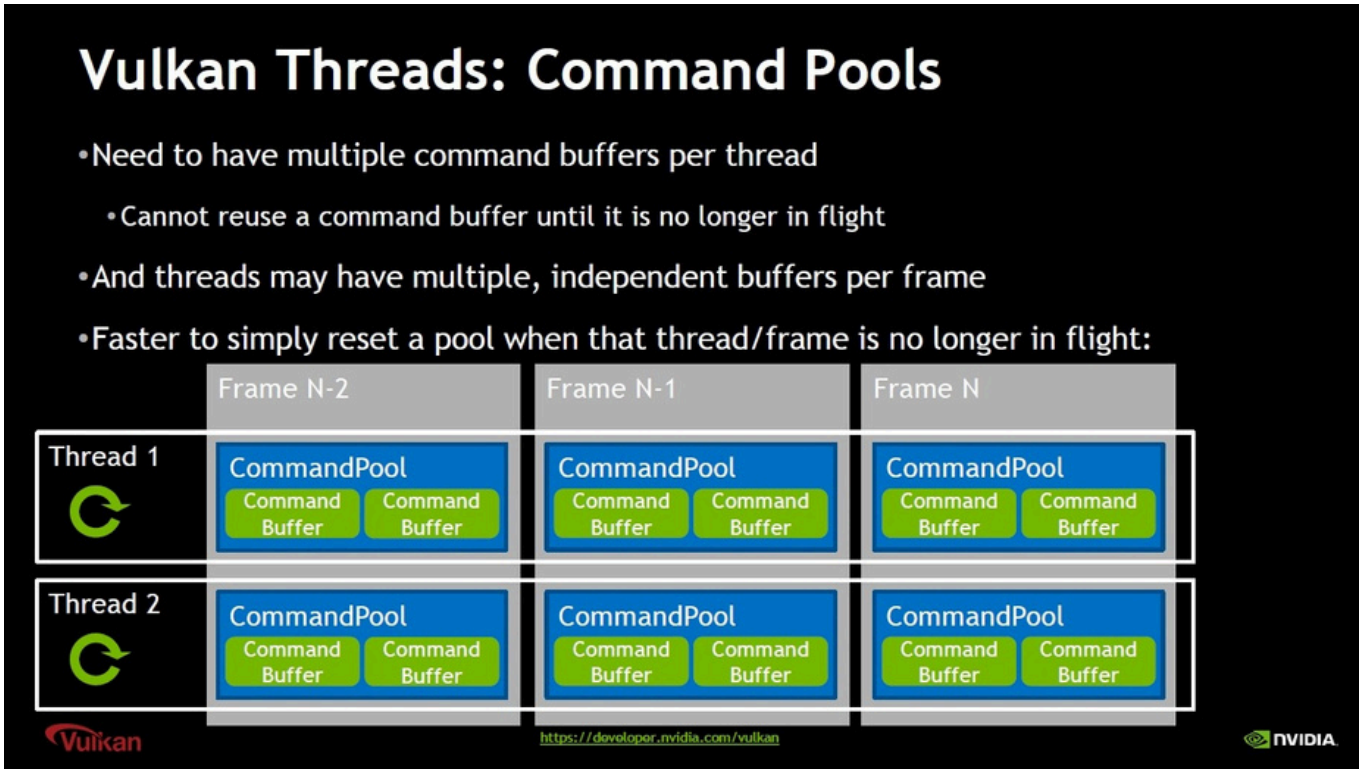
As a new cross-platform graphics API, Vulkan abandons the drawbacks of traditional graphics APIs and faces the advantages of the multi-core era, thus unleashing the power of parallel rendering from the perspective of design and architecture.

In general, Vulkan and DirectX12 are very similar. They both have core concepts such as Command Buffer, CommandPool, Command Queue and Fence. The parallel mode is also very similar: Command Buffer instructions are generated in parallel on different CPU threads, and finally the main thread collects these Command Buffers and submits them to the GPU:



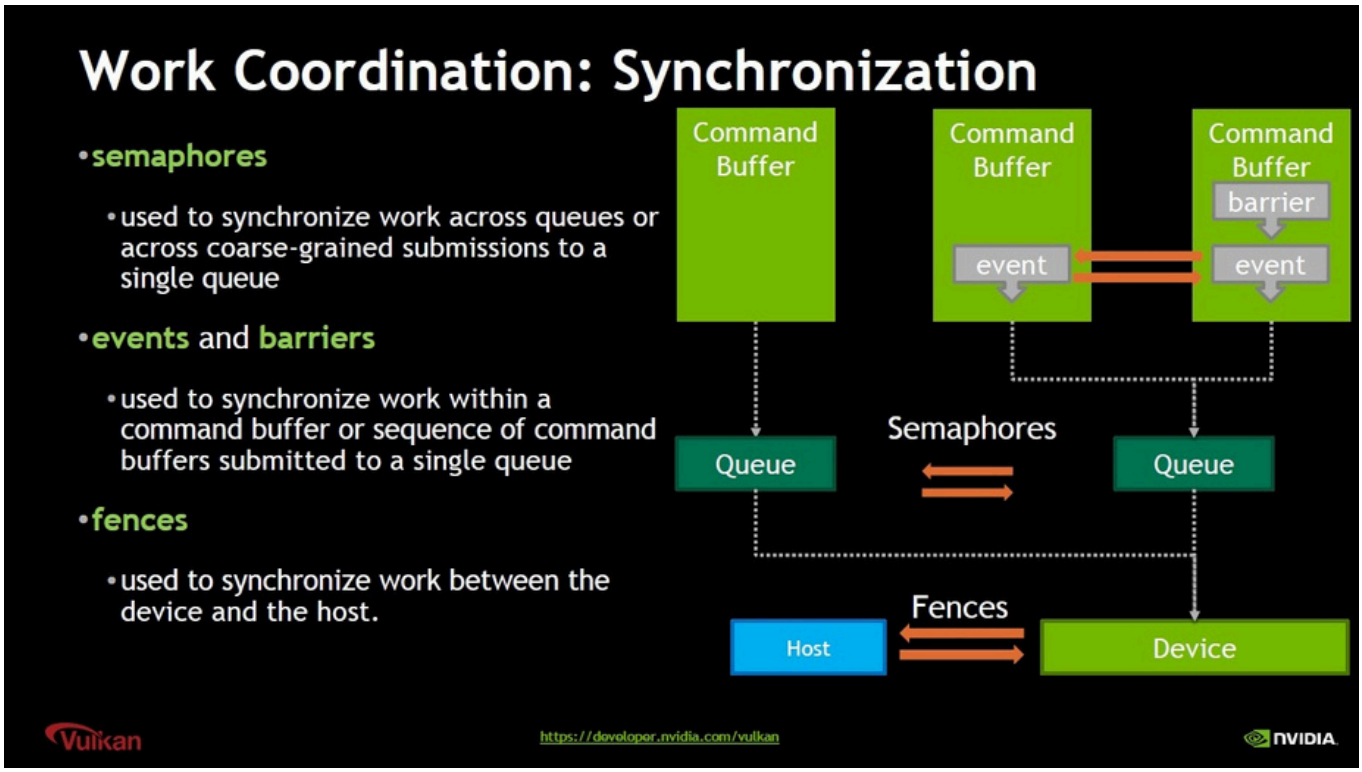
A diagram of the Vulkan graphics API parallelism.

In addition, Vulkan's CommandPool can be created by different threads per frame to reduce synchronization waits and improve parallel efficiency:



Schematic diagram of the parallelization of CommandPool in Vulkan between different frames.

In addition, Vulkan also has various synchronization mechanisms similar to DirectX12:



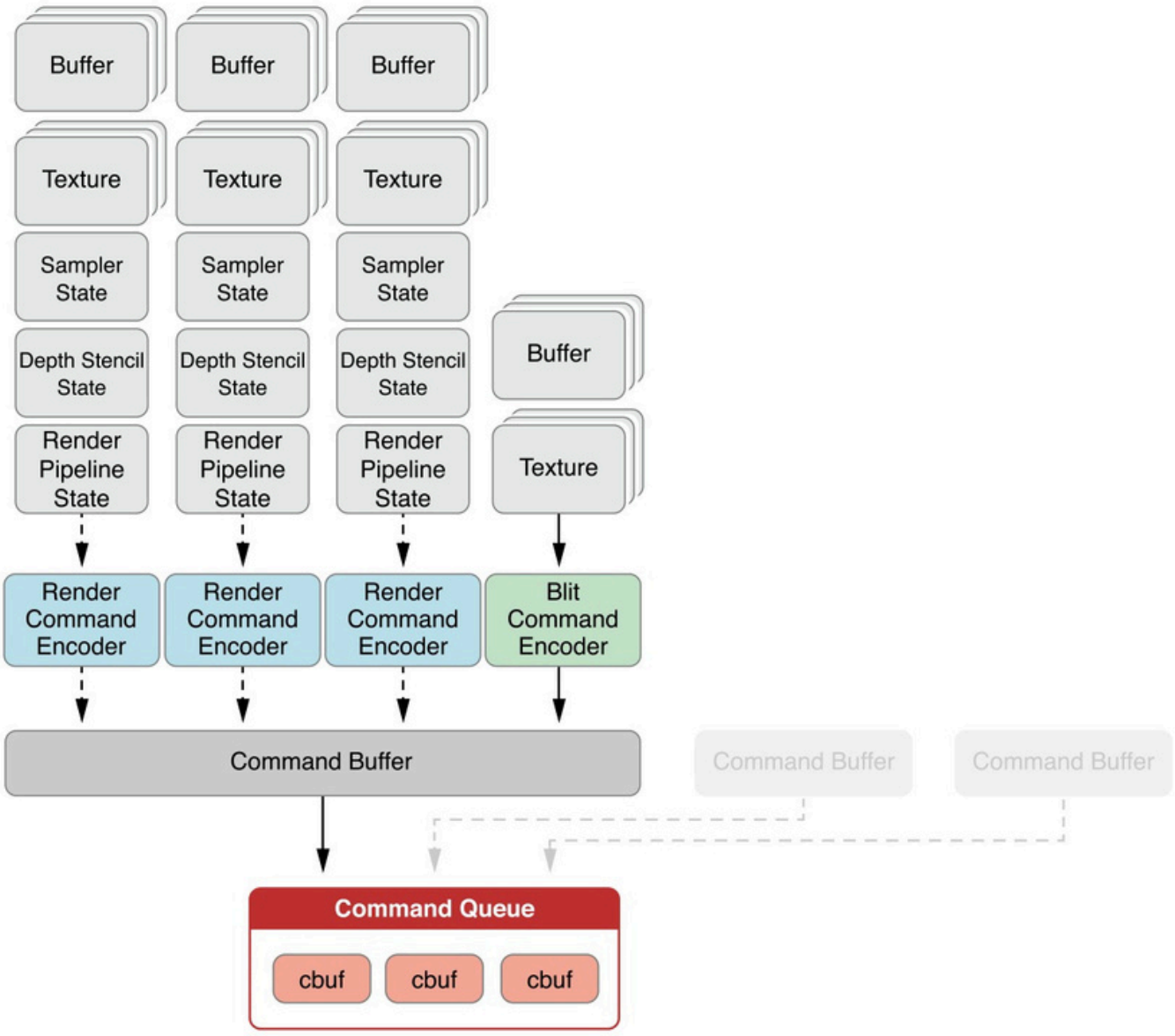
Vulkan synchronization mechanism: semaphore is used to synchronize queues; Fence is used to synchronize GPU and CPU; Event and Barrier are used to synchronize Command Buffer.
For more usage, analysis, and comparison of Vulkan, seeEvaluation of multi-threading in Vulkan .

2.2.4 Metal's Multithreading Features

Metal is the exclusive graphics API for iOS and MacOS systems, and is also a representative of the new generation. It is compatible with traditional graphics API usage such as OpenGL, and also supports the new generation of graphics API concepts and architectures such as Vulkan and DirectX12. From the user's perspective, Metal is more user-friendly, providing an API with a clearer structure and more friendly concepts.

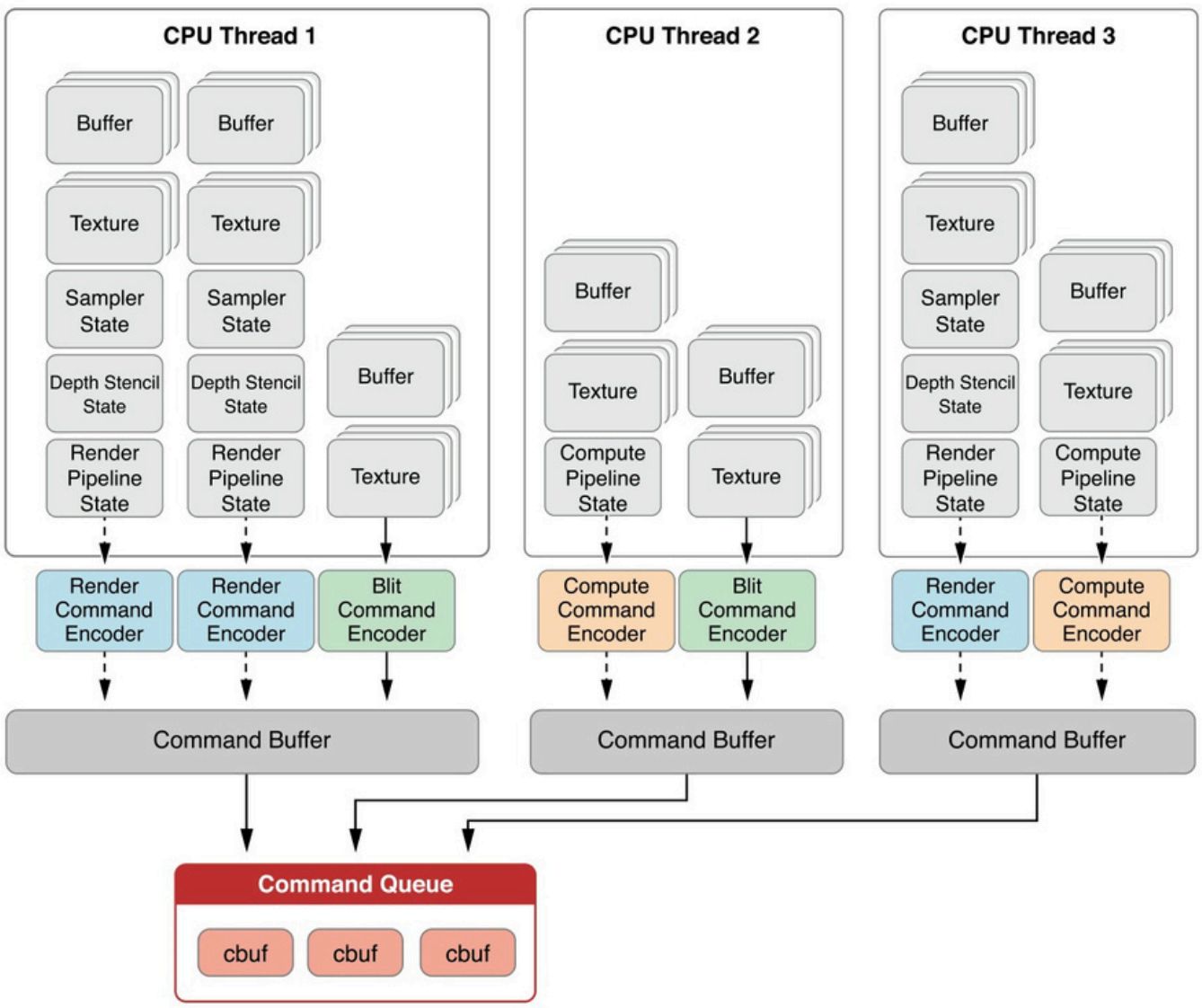
Comparison of the cost and benefits of migrating from OpenGL to new-generation graphics APIs. The horizontal axis is the cost of migrating from OpenGL (or ES) to other graphics APIs, and the vertical axis is the potential performance benefit. It can be seen that Metal's migration cost is lower, but its potential performance ratio is not as high as Vulkan and DirectX12.

Metal, like Vulkan and DirectX, has many similar concepts, such as Command, Command Buffer, Command Queue and various synchronization mechanisms.



A table showing the relationship between the basic concepts of Metal. There are three types of Command Encoders: MTLRenderCommandEncoder, MTLComputeCommandEncoder, and MTLBlitCommandEncoder. After CommandEncoder records the command, it is inserted into the Command Buffer and finally enters the Command Queue.

With similar concepts and mechanisms, Metal can also easily implement multi-threaded recording commands and support multi-threaded scheduling from the hardware level:



Schematic diagram of Metal multithreading model. The figure shows that three CPU threads are recording different types of encoders at the same time. Each thread has its own command buffer. Finally, these command buffers are unified into the command queue and executed by the GPU.

2.3 Multithreaded rendering of game engines

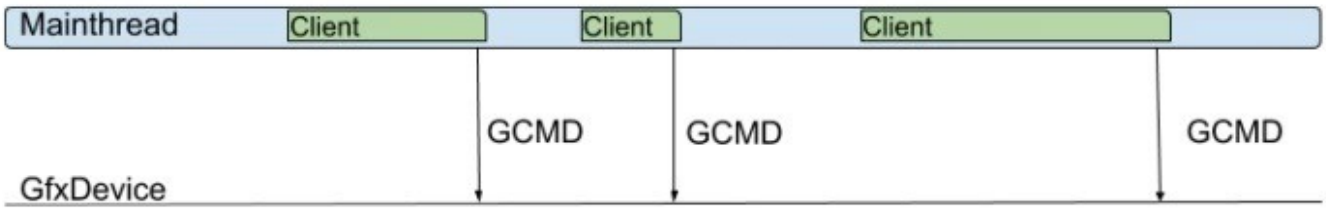
Before formally explaining UE's multi-threaded rendering, let's first understand the multi-threaded architecture and design of other mainstream commercial engines.

2.3.1 Unity

There are several core concepts in Unity's rendering system. One is the Client, which runs on the main thread (logical thread) and is responsible for generating rendering instructions; the other is the Worker Thread, which is used to assist in processing the main thread or generating various sub-tasks such as rendering instructions. Unity's rendering architecture supports the following modes:

- **Singlethreaded Rendering**

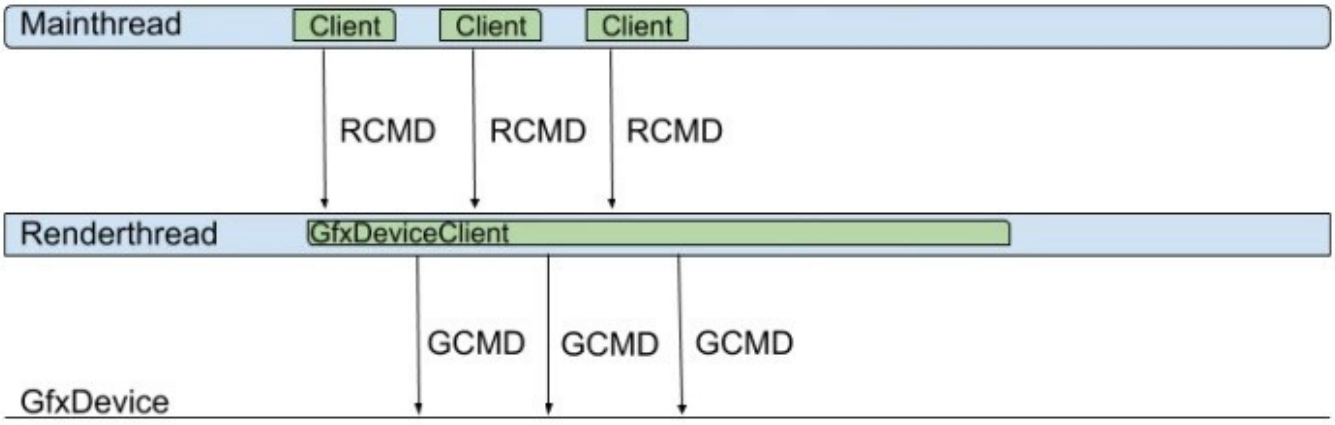
Single-threaded rendering mode, in this mode there is only a single Client component, no worker thread. The only Client generates all rendering commands (RCMD) in the main thread, and has a graphics device object, and also generates graphics API commands (GCMD) to the graphics device in the main thread. Its scheduling diagram is as follows:



In this mode, the CPU and GPU may wait for each other, and the multi-core CPU cannot be fully utilized, resulting in the worst performance.

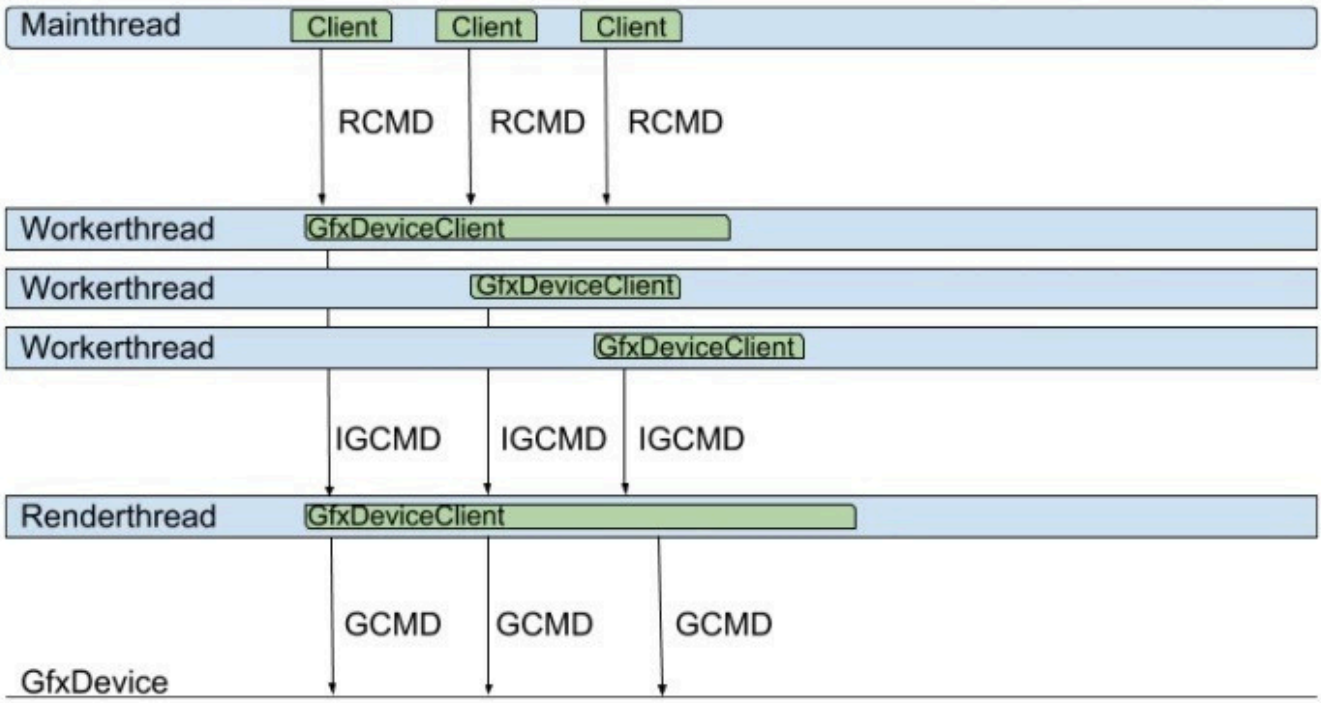
- * * Multithreaded Rendering **

In multi-threaded rendering mode, compared with single-threaded mode, there is one more working thread, namely the rendering thread used to generate GCMD. The rendering thread runs the GfxDeviceClient object, which is dedicated to generating graphics API instructions for the corresponding platform:



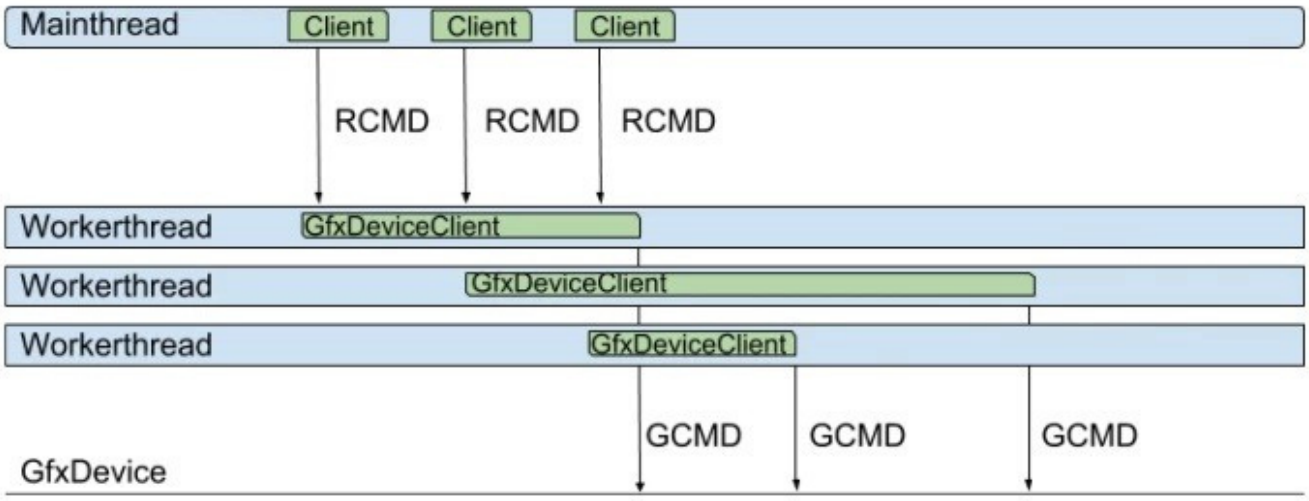
- **Jobified Rendering**

Job rendering mode, in this mode there are multiple Client objects and a single rendering thread. In addition, there are multiple job objects, each of which runs in a dedicated independent thread to generate intermediate graphics commands (IGCMD). In addition, there is a worker thread (rendering thread) used to convert the IGCMD generated by the job thread into the GCMD of the graphics API. The operation diagram is as follows:



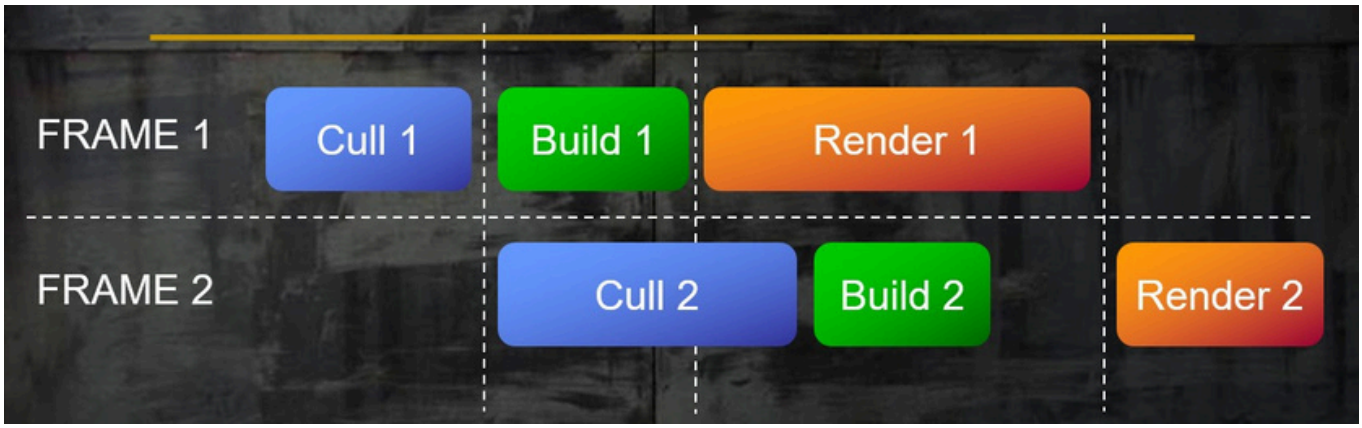
- **Graphics Jobs**

Graphical job rendering mode, in this mode there are multiple Clients, multiple worker threads, and no rendering thread. Multiple Client objects on the main thread drive the corresponding graphics device objects on the worker thread, directly generating GCMD, thereby avoiding the generation of IGCMD intermediate instructions in the Jobified Rendering mode. It can only be enabled on graphics APIs that support hardware-level multithreading, such as DirectX12, Vulkan, etc. The running diagram is as follows:

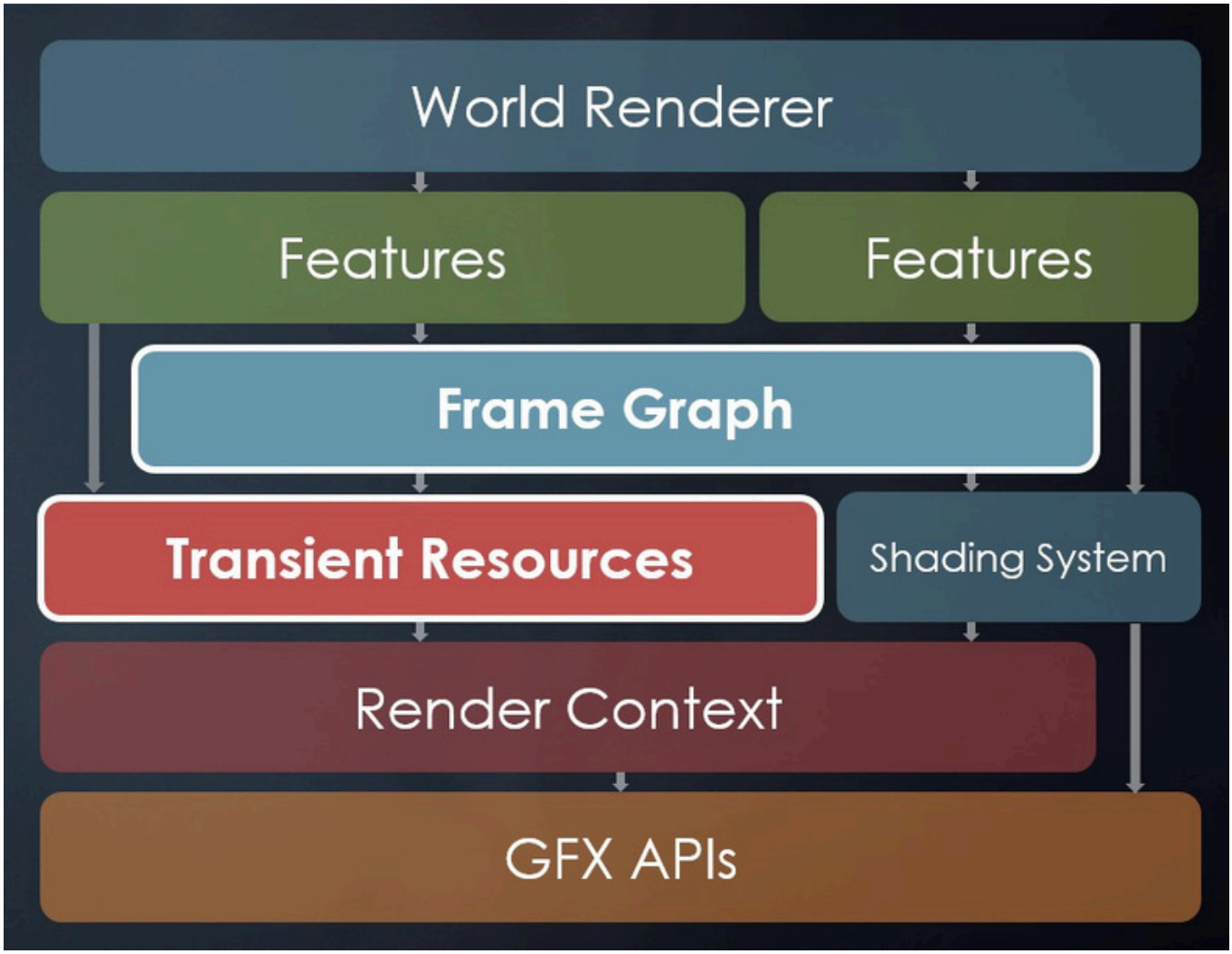


2.3.2 Frostbite

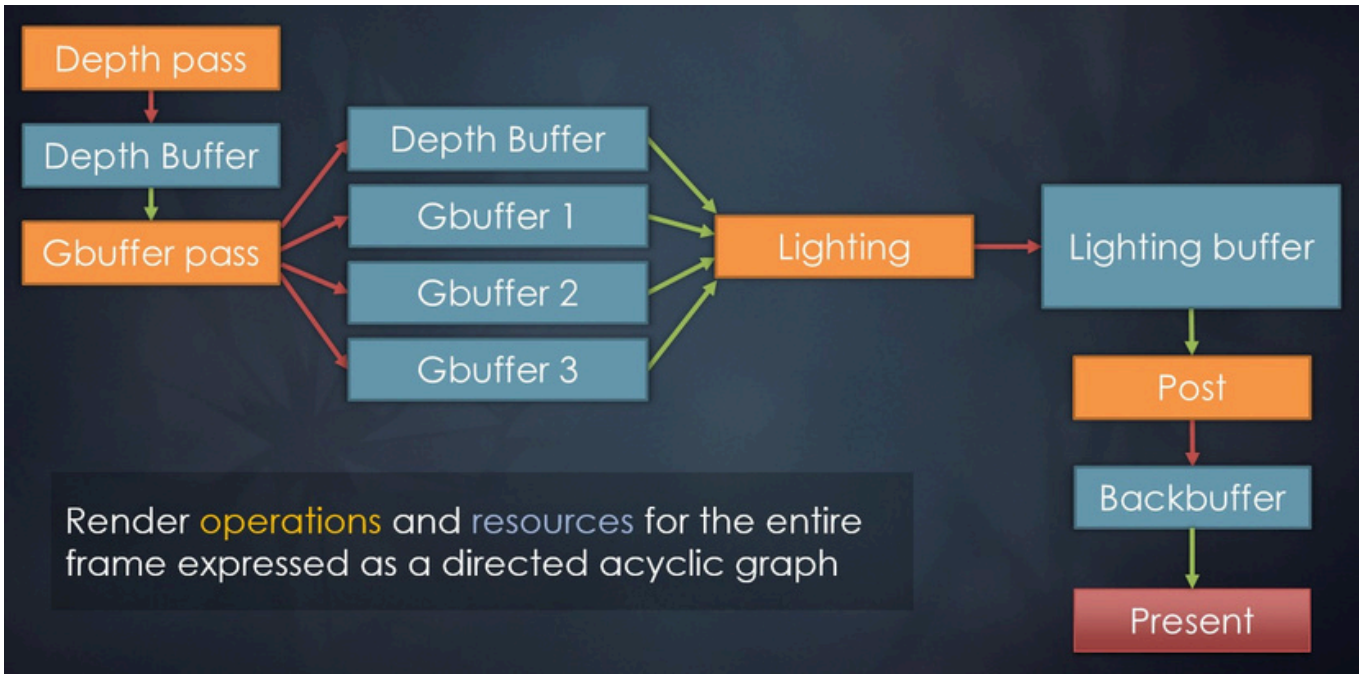
In the early days of the Frostbite engine, each frame was divided into three steps: cropping, building, and rendering. The data required for each step was placed in a double buffer, and it was run in a cascade manner, using a simple synchronization process. Its operation diagram is as follows:



After years of evolution, Frostbite adopted the multi-threaded rendering mode of Frame Graph in the past few years. This mode is designed to isolate the engine's various rendering functions (Features) from the upper rendering logic (Renderer) and lower resources (Shader, RenderContext, graphics API, etc.) for further decoupling and optimization, the most important of which is to enable multi-threaded rendering.



FrameGraph is a high-level representation of Render Passes and resources, containing all the information used in a frame. The order and dependencies between Passes can be specified, and the following figure is an example:



The order and dependency graph of deferred rendering implemented by Frostbite Engine using frame graph method.

Each frame of the frame graph has three stages: Setup, Compile and Execute.

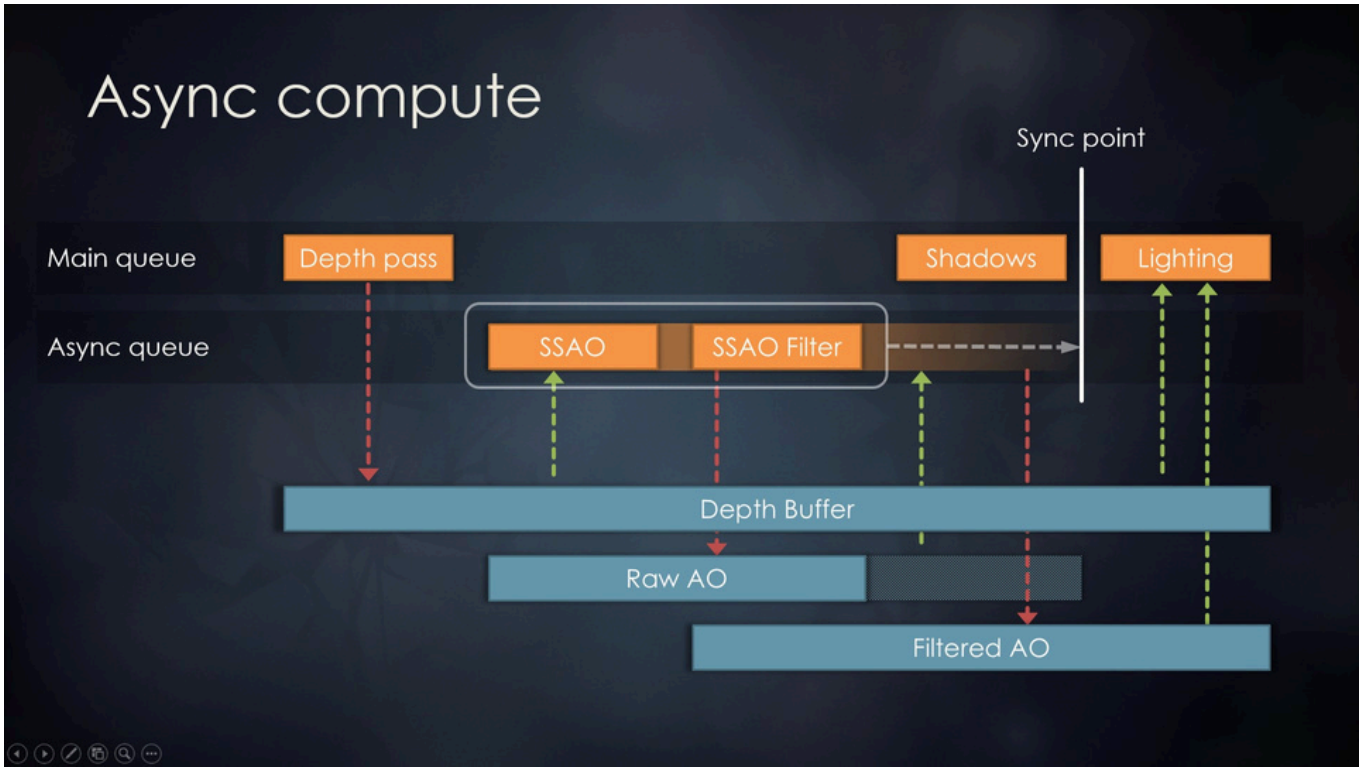
The establishment phase is to create information such as various Render Passes, input textures, output textures, dependent resources, etc.
The work of the compilation phase is mainly to remove unused Render Passes and resources, calculate resource lifecycles, and create corresponding GPU resources based on usage tags. A lot of optimizations are done when creating GPU resources, such as: simplifying the video memory allocation algorithm, applying for the first use and releasing it after the last use, asynchronously calculating the lifecycle of external resources, precise resource management from binding tags, flattening all resource references to improve the GPU cache hit rate, etc. The compilation phase uses linear traversal of all RenderPasses, and during the traversal, it calculates the number of resource references, the first and last users of resources, asynchronous waiting points, and resource barriers, etc.

The execution phase is executed in the order of Setup (the compilation phase will not be reordered), and only those Render Passes that have not been eliminated are traversed and their callback functions are executed. If it is immediate mode, the device context API is called directly. The execution phase will actually obtain GPU resources according to the handle generated in the compilation phase.

The most important thing is that the whole process realizes automated asynchronous calculation through the dependency graph. The asynchronous mechanism starts at the main timeline, automatically synchronizing resources in different queues, and extending their lifecycles to prevent accidental release. Of course, this automated system also has side effects, such as adding a certain amount of additional memory, which may cause unexpected performance bottlenecks. Therefore,

the Frostbite Engine supports manual mode to control and change the asynchronous operation mode as expected, so as to selectively join each Render Pass.

The following figure can explain the operation mechanism of asynchronous computing in a concise and clear manner:



Schematic diagram of asynchronous calculation of Frost Engine. The Pass of SSAO and SSAO Filter are placed in the asynchronous queue, which will write and read the texture of Raw AO. Even if it ends before the synchronization point, the life cycle of Raw AO will still be extended to the synchronization point.

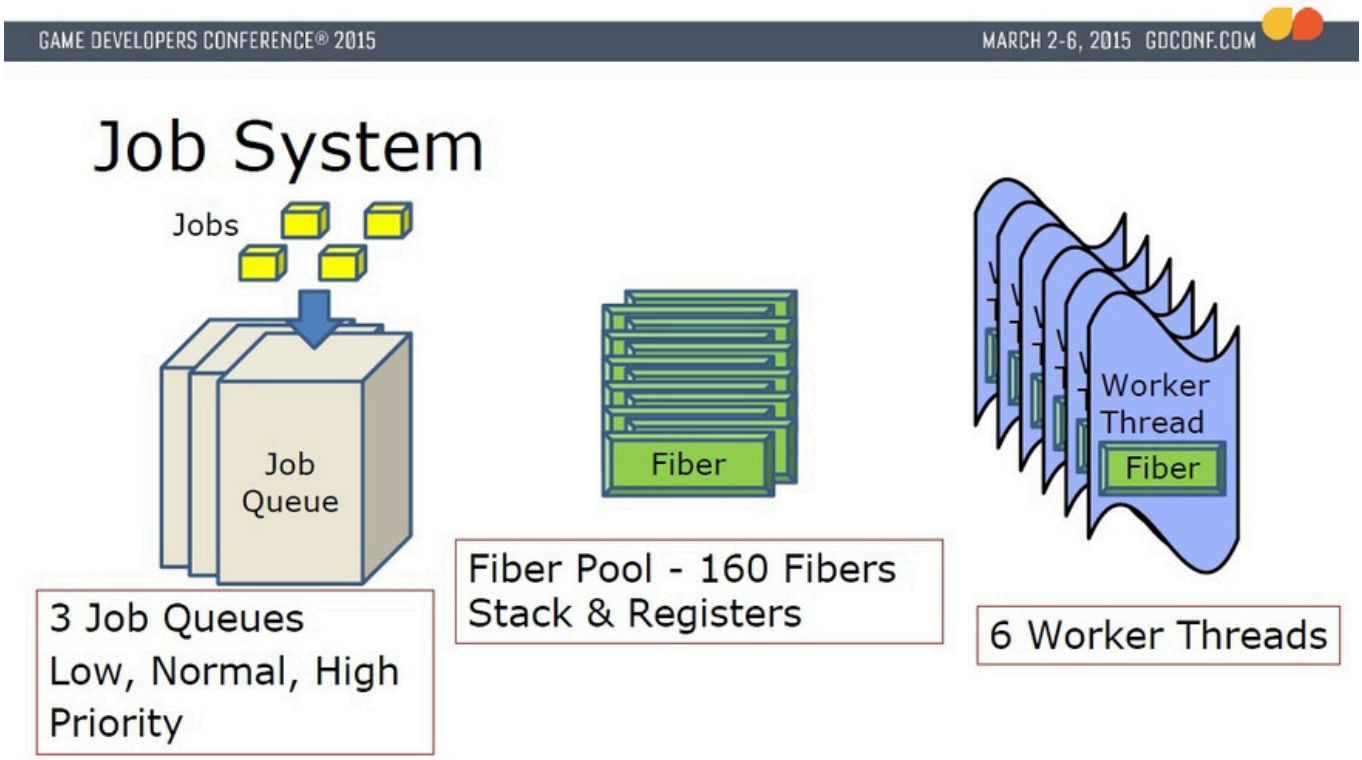
In short, the frame graph rendering architecture benefits from recording all the information of the frame, so that a large amount of memory and video memory can be saved through resource aliasing, semi-automatic asynchronous calculation can be achieved, rendering pipeline control can be simplified , and better visualization and diagnostic tools can be produced.

2.3.3 Naughty Dog Engine

Naughty Dog's game engine also uses a job system, which allows non-GPU logic code to be added to the job system. Jobs can directly start and wait for other jobs, hiding memory management details from the caller, providing a simple and easy-to -use API, and putting performance optimization in the second place.

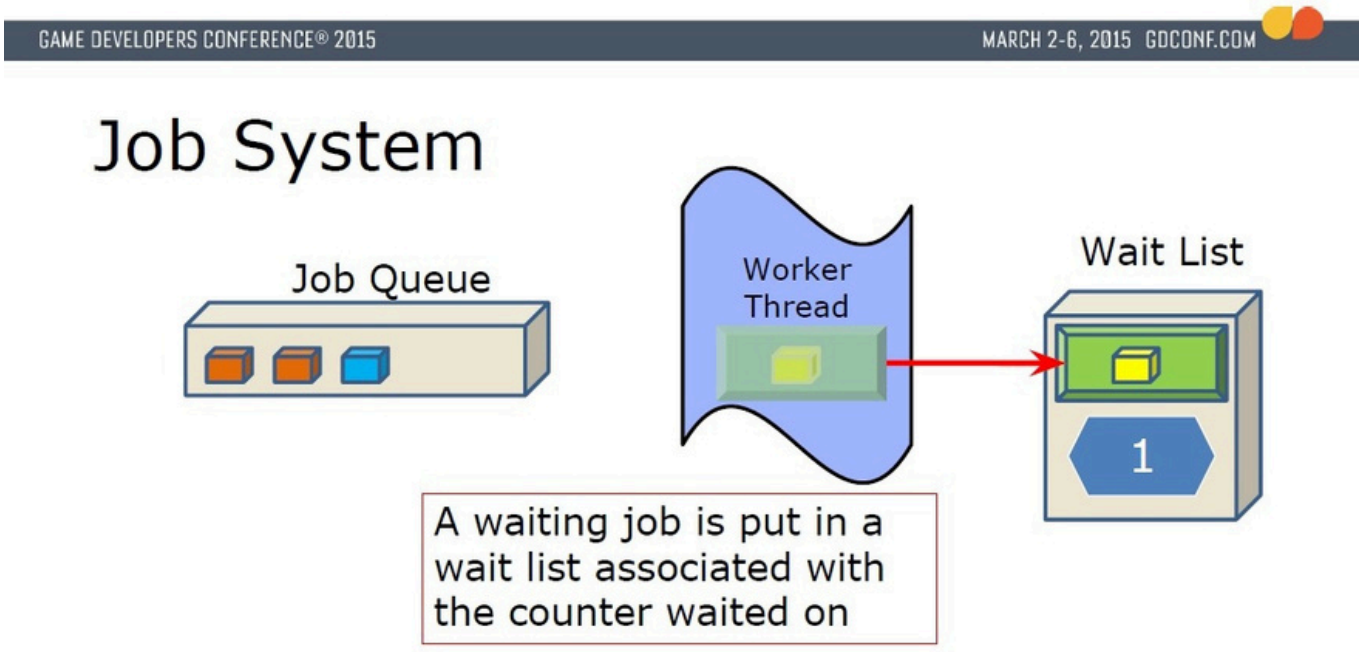
The job system runs on fibers. Each fiber is similar to a local thread. The user layer provides stack space, and its context contains a small amount of fiber state to reduce register occupancy. It actually runs on threads, a cooperative multi-threading model . Since fibers are not system-level threads, context switching is very fast, and only the register state (program counter, stack pointer, gpr, etc.) is saved and restored, so the overhead is very small.

The operating system will open up several working threads, each of which will be locked to the GPU hardware core. Threads are execution units, fibers are contexts, and operations are always executed within the context of threads, using atomic counters for synchronization. The following figure is the operating system architecture of the Naughty Dog engine:



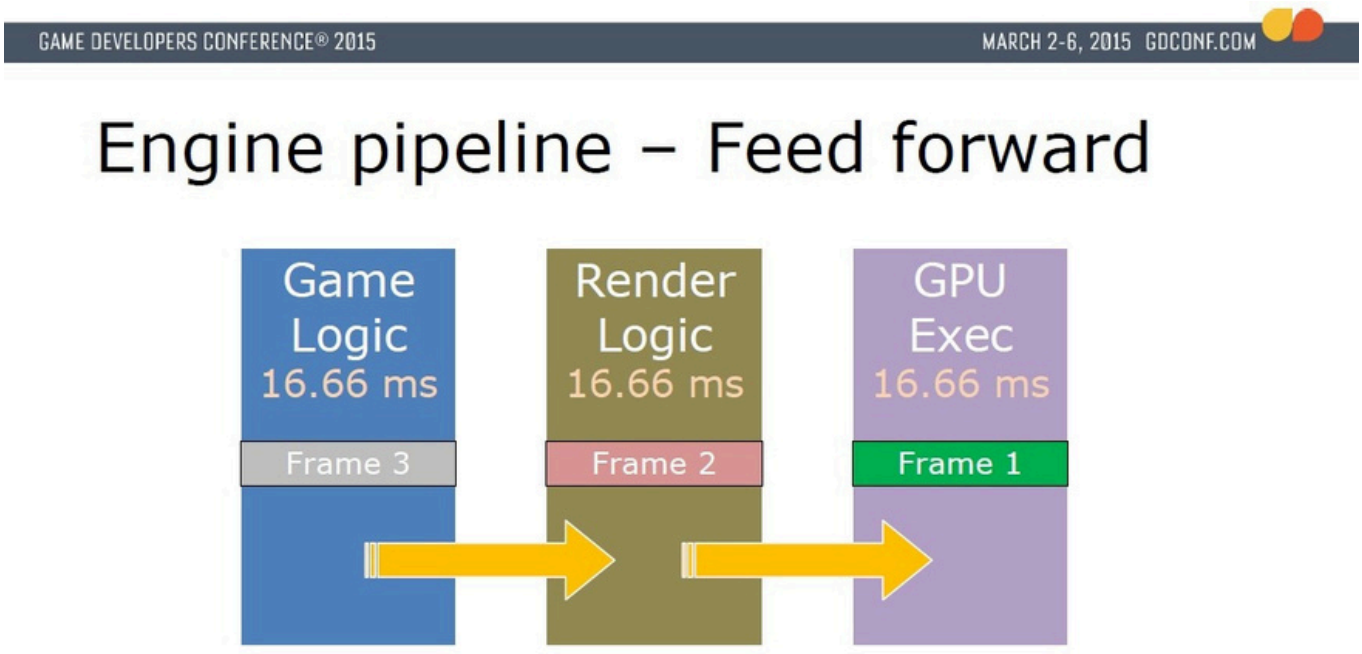
Naughty Dog engine job system architecture diagram. It has 6 worker threads, 160 fibers, and 3 job queues.

Jobs can add new jobs to the job queue, and waiting jobs will be placed in a special waiting list. Each waiting job will have a reference count, and will not be removed from the waiting queue until the reference count reaches 0 for continued execution.



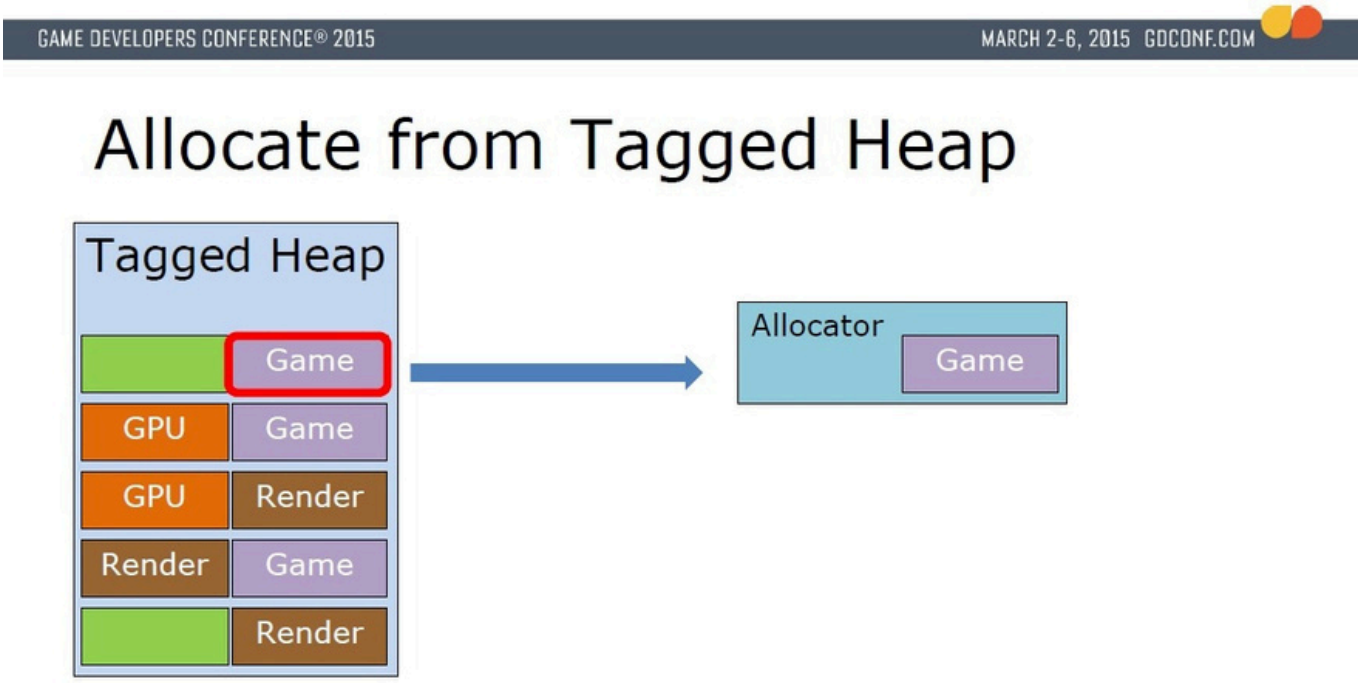
In the Naughty Dog engine, everything except IO threads is a job, including game object updates, action updates and blending, ray detection, rendering command generation, etc. It can be seen that the job system is fully utilized to maximize the proportion and efficiency of parallelism.

In order to improve the frame rate, the game logic and rendering logic are separated and executed in parallel, but the data of different frames are processed. Usually, the game data is one frame ahead of the rendering data, and the rendering logic is one frame ahead of the GPU data.

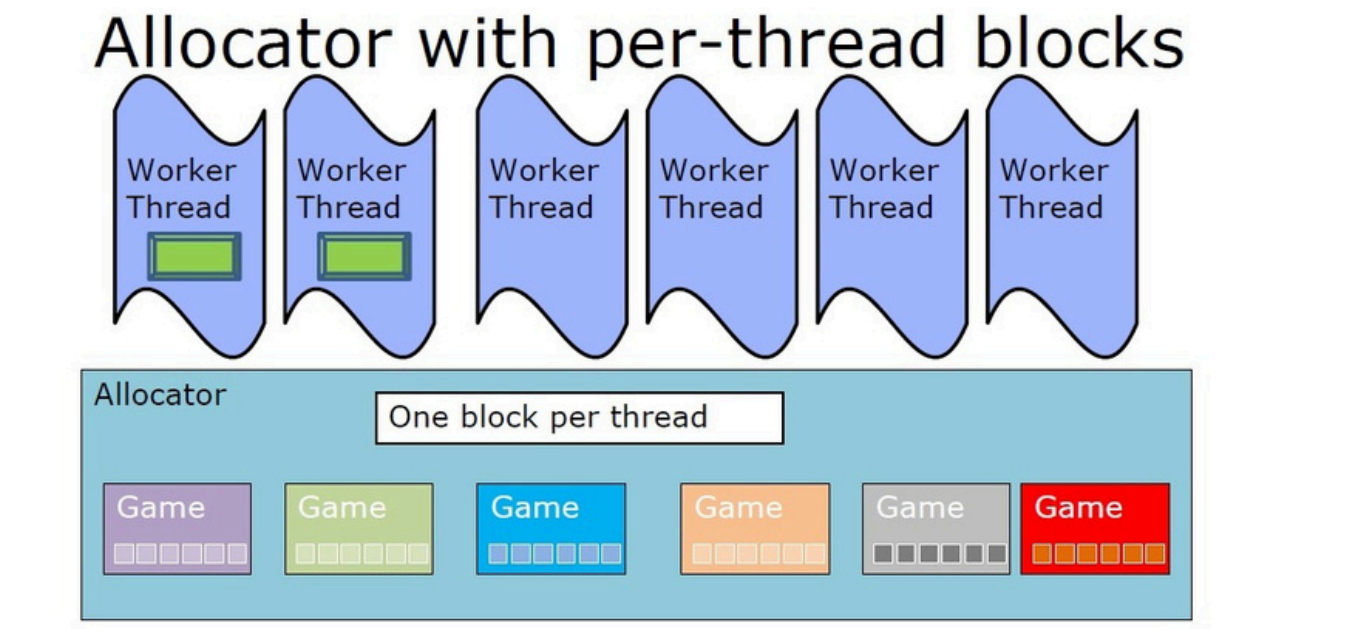


Through such a mechanism, synchronization and waiting between CPU threads and between the CPU and GPU can be avoided, thereby improving frame rate and throughput.

In addition, its memory allocation is also managed in a sophisticated way. For example, a tagged heap is introduced. The memory heap is a 2M block. Each block has a tag (one of Game, Render, and GPU). The allocator allocates and releases memory in the tagged heap to avoid directly obtaining it from the operating system:



In addition, the allocator supports allocating a dedicated block to each worker thread (similar to TLS), avoiding data synchronization and waiting time, and avoiding data contention.

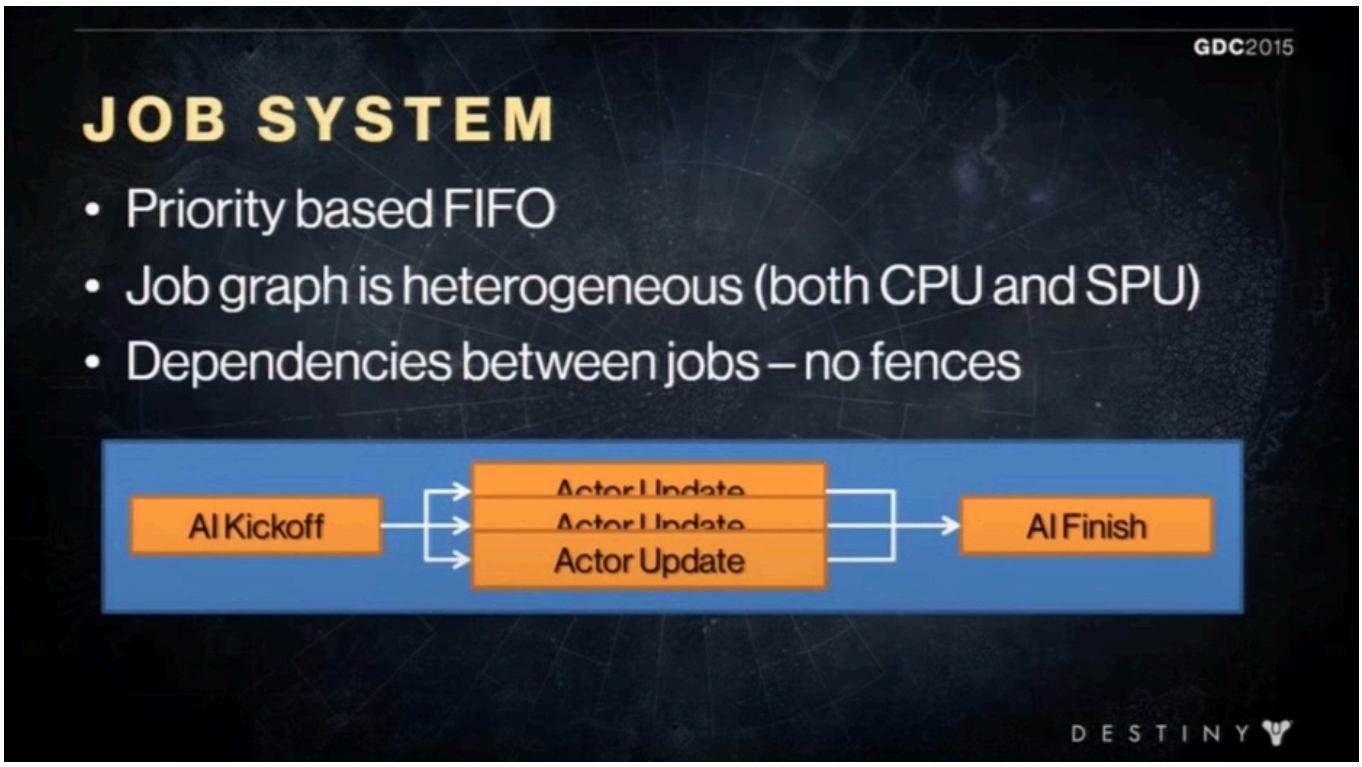


2.3.4 Destiny's Engine

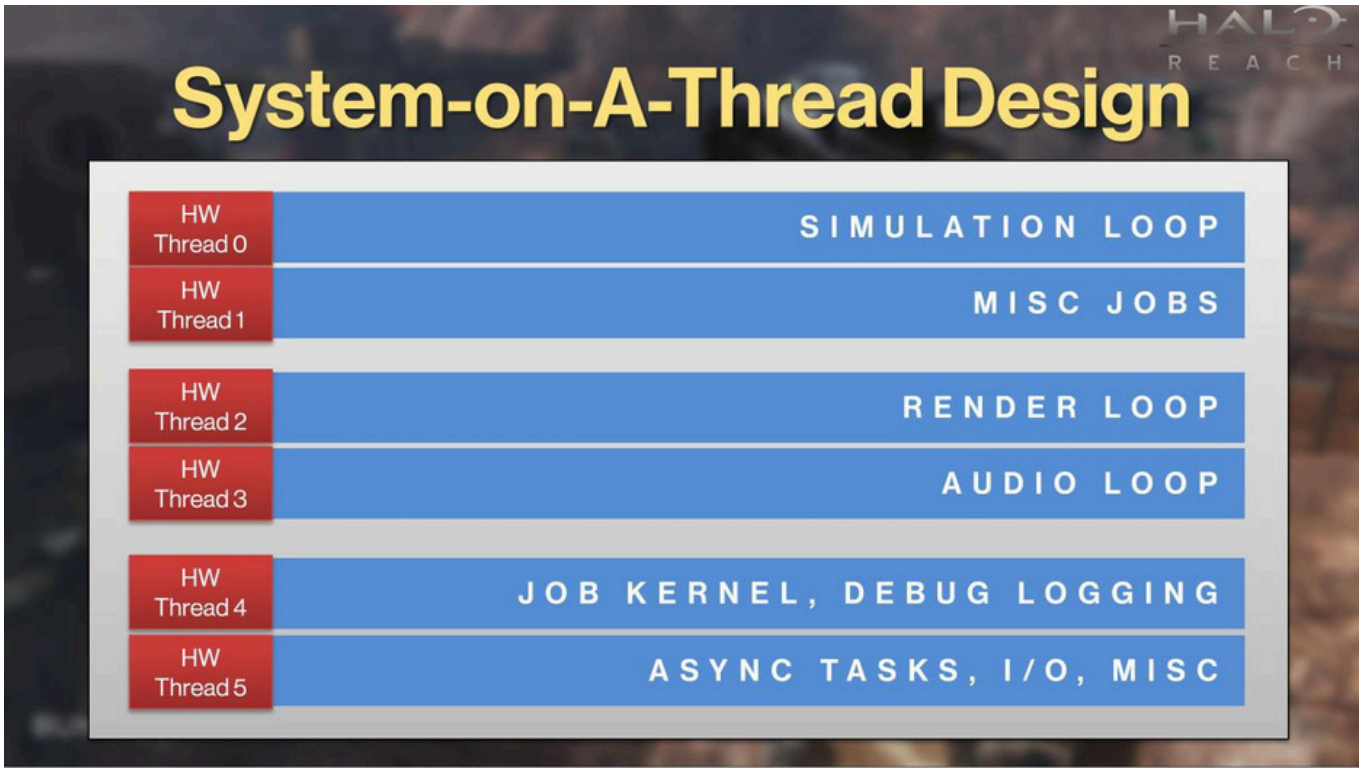
Destiny is a first-person action role-playing MMORPG, and the engine it uses is also called Destiny's Engine.

Destiny Engine uses multi-threaded architecture, task-based parallel processing, job management design and synchronous processing, and job execution is also on fibers. The priority of job execution

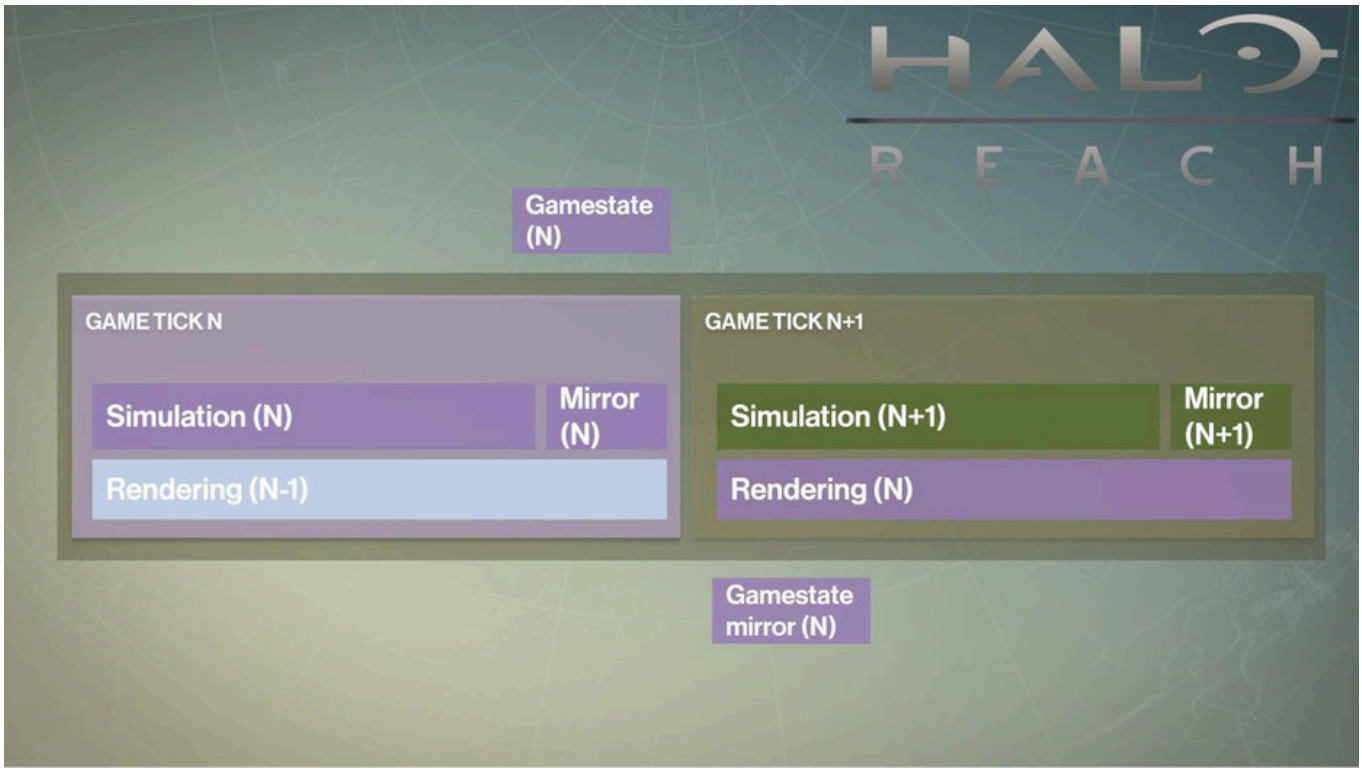
in the job system is FIFO (first in, first out), the job graph is a heterogeneous architecture, and there are dependencies between jobs, but no fences.



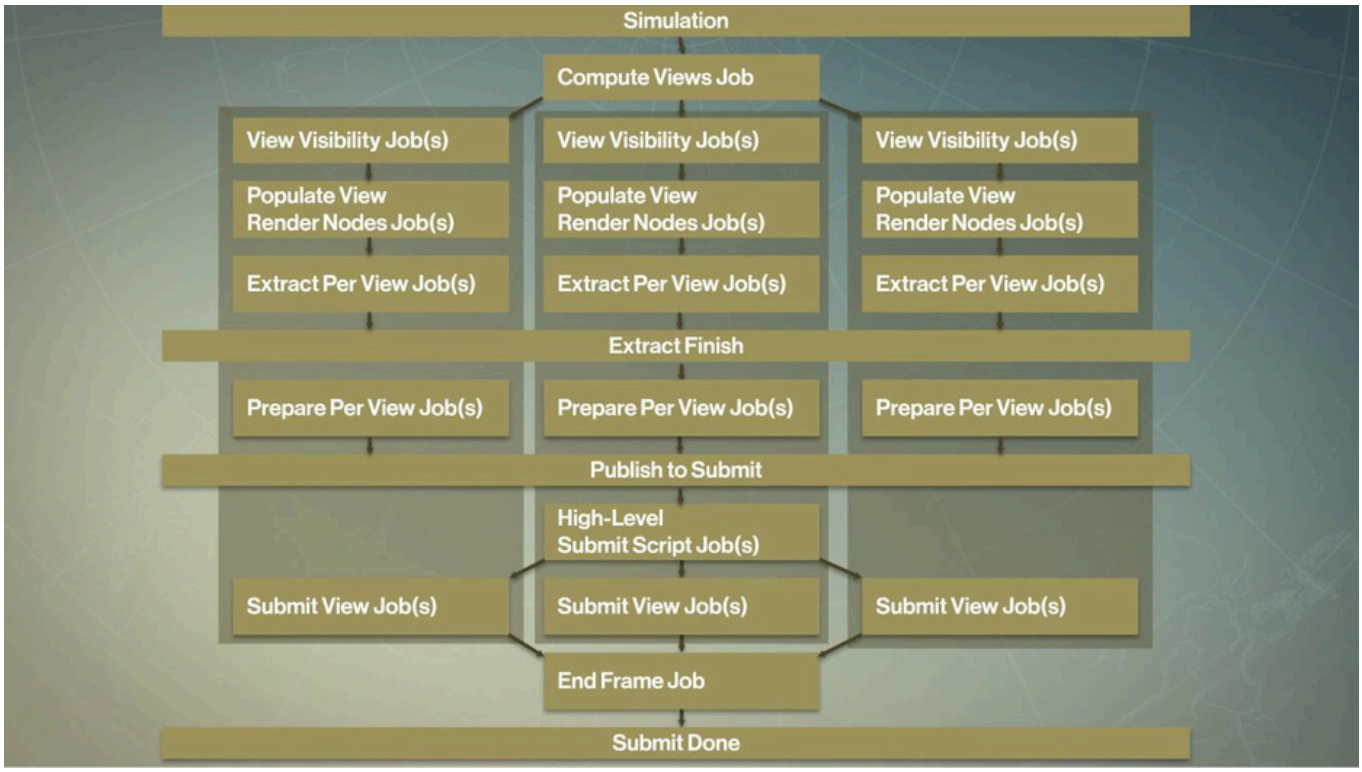
It divides each frame into several steps: simulating game objects, object clipping, generating rendering commands, executing GPU-related work, and displaying. In terms of thread design, 6 system threads will be created, and the content of each thread is: simulation loop, other jobs, rendering loop, audio loop, job core and debug log, asynchronous tasks, IO, etc.



When processing data between frames, the game simulation and rendering logic are also separated. The game simulation is always one frame ahead of the rendering. After the game simulation is completed, all data and status will be copied (mirrored) for use in the rendering of the next frame:



In order to maximize the parallel efficiency of CPU and GPU, Destiny Engine adopts dynamic load balancing and smart job batching. The specific approach is to add all rendering and visibility culling work to the task system to maintain low latency. The following figure is an example of parallel computing view job:



In addition, the simulation logic is extracted and decoupled from the rendering logic, and a fully data-driven rendering pipeline is adopted. All sorting, memory allocation, traversal and other algorithms follow cache consistency (structure minimization and data alignment so that a single structure data can be loaded into the cache line at one time).

2.4 UE's multi-threading mechanism

This chapter mainly the multi-threaded foundation, design and architecture analysis of UE, so as to better enter the multi-threaded rendering later.

2.4.1 UE's multithreading foundation

- **TAtomic**

UE's atomic operations do not use C++'s Atomic template, but implement a set of its own, called TAtomic. It provides functions such as loading, storing, and assigning operations. In the underlying implementation, it will use the platform-related atomic operation interface:

```
// Engine\Source\Runtime\Core\Public\Templates\Atomic.h

template <typename T>
FORCEINLINE TLoad(constvolatileT* Element) {

	//Use platform-dependent interfaces to load atomic values.
	autoResult = FPlatformAtomics::AtomicRead((volatile
	TUnderlyingIntegerType_T<T>*)Element);
	return*(constT*)&Result;
}

template <typename T>
FORCEINLINEvoid Store(constvolatileT* Element, T Value) {

	//Use platform-dependent interfaces to store atomic values.
	FPlatformAtomics::InterlockedExchange((volatileTUnderlyingIntegerType_T<T>*)Element,
* (constTUnderlyingIntegerType_T<T>*)&Value); }


template <typename T>
FORCEINLINE TExchange(volatileT* Element, T Value) {

	//Use platform-dependent interfaces to exchange atomic values.
	autoResult = FPlatformAtomics::InterlockedExchange((volatile TUnderlyingIntegerType_T<T>*)Element, *(
constTUnderlyingIntegerType_T<T>*)&Value);
	return*(constT*)&Result;
}
```

In terms of memory order, unlike C++ which provides four modes, UE simplifies it and only provides two modes:

```
enum classEMemoryOrder {

	Relaxed,		//Loose order,	No reordering
	SequentiallyConsistent	//Sequential consistency
};
```

It should be noted that although TAtomic is a template class, it only works for basic types. UE uses the parent classTAtomicBaseType_Tto achieve the purpose of detection:

```
template <typename T>
classTAtomicfinal: public UE4Atomic_Private::TAtomicBaseType_T<T> {

	static_assert(TIsTrivial<T>::Value,"TAtomic is only usable with trivial types");

	(.....)
}
```

- **TFuture**

UE implements Future and Promise objects similar to C++. It is a template class that abstracts the return value type. The following is the declaration of TFuture:

```
// Engine\Source\Runtime\Core\Public\Async\Future.h

template<typename	InternalResultType>
classTFutureBase
{
public:
	bool	IsReady()	const;
	bool	IsValid()	const;

	void Wait()const {

		if(State.IsValid())
		{
			while(!WaitFor(FTimespan::MaxValue()));
		}
	}
	bool WaitFor(constFTimespan& Duration)const {

		returnState.IsValid() ? State->WaitFor(Duration) :false;
	}
	bool WaitUntil(constFDateTime& Time)const {

		returnWaitFor(Time - FDateTime::UtcNow());
	}

protected:
	typedefTSharedPtr<TFutureState<InternalResultType>, ESPMode::ThreadSafe> StateType;

	constStateType&GetState()const;

	template<typename Func> autoThen
	(Func Continuation);

	template<typename Func> autoNext
	(Func Continuation);

	void	Reset();

private:
```

```
    /** Holds the future's state. */ StateType
    State;
};

template<typename ResultType>
classTFuture: public TFutureBase<ResultType> {

    typedefTFutureBase<ResultType> BaseType;

public:

    ResultType    Get()const
    {
        return    this->GetState()->GetResult();
    }

    TSharedFuture<ResultType>Share() {

        returnTSharedFuture<ResultType>(MoveTemp(*this));
    }
};
```

- **TPromise**

TPromise is usually used with TFuture, as shown below:

```
template<typename InternalResultType> class
TPromiseBase: FNoncopyable {

    typedefTSharedPtr<TFutureState<InternalResultType>, ESPMode::ThreadSafe> StateType;

    (.....)

protected:
    constStateType&GetState();

private:
    StateType State;//StoredFutureStatus.
};

template<typename ResultType>
classTPromise: public TPromiseBase<ResultType> {

public:
    typedefTPromiseBase<ResultType> BaseType;

public:
    //GetFutureObject
    TFuture<ResultType>        GetFuture()
    {
        check(!FutureRetrieved);
        FutureRetrieved =true;

        returnTFuture<ResultType>(this->GetState());
    }
```

```
    //set upFutureValue
    FORCEINLINEvoid SetValue(constResultType& Result) {

        EmplaceValue(Result);
    }

    FORCEINLINEvoid SetValue(ResultType&& Result) {

        EmplaceValue(MoveTemp(Result));
    }

    template<typename...ArgTypes> void
    EmplaceValue(ArgTypes&& ... Args) {

        this->GetState()->EmplaceResult(Forward<ArgTypes>(Args)...);
    }

private:
    bool    FutureRetrieved;
};
```

- **ParallelFor**

ParallelForis a built-in For loop in UE that supports multi-threaded parallel processing tasks. It is widely used in rendering systems. It supports the following parallel methods:

```
enum class      EParallelForFlags
{
    None,    //Default parallel mode
    ForceSingleThread=1,//Force single thread, often used for debugging. Unbalanced =2,//Non-task
    balancing, often used for tasks with highly variable computation times.
    PumpRenderingThread =4,//Inject into the rendering thread. If called in the rendering thread, you need to ensureProcessThreadIdle state.
};
```

The supported ParallelFor calling methods are as follows:

```
inlinevoid ParallelFor(int32 Num, TFunctionRef<void(int32)> Body,bool bForceSingleThread,bool
bPumpRenderingThread=false);

inlinevoid ParallelFor(int32 Num, TFunctionRef<void(int32)> Body, EParallelForFlags Flags = EParallelForFlags::None);


template<typename FunctionType>
inlinevoid ParallelForTemplate(int32 Num,constFunctionType& Body, EParallelForFlags Flags =
EParallelForFlags::None);

inlinevoid ParallelForWithPreWork(int32 Num, TFunctionRef<void(int32)> Body, TFunctionRef<void()>
CurrentThreadWorkToDoBeforeHelping,boolbForceSingleThread,bool bPumpRenderingThread =false);


inlinevoid ParallelForWithPreWork(int32 Num, TFunctionRef<void(int32)> Body, TFunctionRef<void()>
CurrentThreadWorkToDoBeforeHelping, EParallelForFlags Flags = EParallelForFlags::None);
```

ParallelFor is implemented based on the TaskGraph mechanism. Since TaskGraph will be mentioned later, its implementation will not be discussed here. The following is an application case of UE:

```
// Engine\Source\Runtime\Engine\Private\Components\ActorComponent.cpp

//Increased parallelismPrimitiveTo the use case of the scenario.
void FRegisterComponentContext::Process() {

	FSceneInterface* Scene = World->Scene;
	ParallelFor(AddPrimitiveBatches.Num(),                    //quantity
		[&](int32 Index)//callback function,IndexBack to Index {

			if(!AddPrimitiveBatches[Index]->IsPendingKill()) {

				Scene->AddPrimitive(AddPrimitiveBatches[Index]);
			}
		},
		!FApp::ShouldUseThreadingForPerformance()//Whether to multi-thread
	) ;
	AddPrimitiveBatches.Empty();
}
```

- **Basic template**

UnrealTemplate.h defines many basic templates for data conversion, copying, transfer, etc. Here are some common functions and types:

Template Name	Analysis	stl mapping
template ReferencedType* IfAThenAElseB(ReferencedType* A,ReferencedType* B)	Return A ? A : B	-
template void Move(T& A,typename TMoveSupportTraits::Copy B)	Release A and replace B's data with A, but it will not affect B's data.	-
template void Move(T& A,typename	Release A and replace B's data with A, but it will affect B's data.	-

Template Name	Analysis	stl mapping
TMoveSupportTraits::Move B)		
FNoncopyable	Deriving from it can implement non-copyable objects.	-
TGuardValue	For values with a job scope, you can specify a new value and an old value. The new value is within the scope and the old value is returned when it leaves the scope.	-
TScopeCounter	A counter with a scope. The counter is +1 within the scope and -1 after leaving the scope.	-
template typename TRemoveReference::Type && MoveTemp(T&& Obj)	Converts a reference to an rvalue, possibly modifying the source value.	std::move
template T CopyTemp(T& Val)	Forces creation of a copy of the rvalue, without changing the source value.	-
template T&& Forward(typename TRemoveReference::Type & Object)	Converts a reference to an rvalue reference.	std::forward
template <typename T, typename ArgType> T StaticCast(ArgType&& Arg)	Static type conversion.	static_cast

2.4.2 UE multi-threading implementation

UE's multi-threaded implementation did not adopt the C++11 standard library, but encapsulated and implemented it at the system level, including system threads, thread pools, asynchronous tasks, task graphs, and related notification and synchronization mechanisms.

2.4.2.1 FRunnable

FRunnableIt is the parent class of all objects that can run in parallel in multiple threads. The basic interface it provides is as follows:

```
// Engine\Source\Runtime\Core\Public\HAL\Runnable.h

classCORE_API        FRunnable
{
public:
        virtual     bool Init();              //initialization, Successful returnTrue.
        virtual     uint32     Run();          //    run,  onlyInitIt will be called only if successful.
        virtual     void Stop();              // Request early stopping.
        virtual     void Exit();              // Exit and clean up the data.
};
```

FRunnableand its subclasses are objects that can run in multiple threads, as opposed to classes that only run in a single threadFSingleThreadRunnable:

```
// Engine\Source\Runtime\Core\Public\Misc\SingleThreadRunnable.h

//Objects running in a single thread with multithreading disabled
classCORE_API FSingleThreadRunnable {

public:
        virtualvoid Tick();
};
```

FRunnableThere are many subclasses of , the following are some common core subclasses and their analysis.

- **FRenderingThread**:An object that runs on the rendering thread. This will be discussed in detail in the following chapters.
- **FRHIThread**:An object that runs on the RHI thread. This will be discussed in detail in the following chapters.
- **FRenderingThreadTickHeartbeat**:Object running on the heartbeat rendering thread.
- **FTaskThreadBase**:The parent class of tasks executed in threads. There will be a chapter
- dedicated to analyzing this part later.
- **FQueuedThread**:The parent class of threads that can be stored in the thread pool. The
- provided interfaces are as follows:

```
// Engine\Source\Runtime\Core\Private\HAL\ThreadingBase.cpp

classFQueuedThread: public FRunnable {

protected:
        FEvent* DoWorkEvent;//Task completed event. TAtomic<bool>
        TimeToDie;//Whether a timeout is required. IQueuedWork*volatile
        QueuedWork;//The task being performed.
        classFQueuedThreadPoolBase*OwningThreadPool;//The thread pool where it is located.
        FRunnableThread* Thread;//The thread that is actually used to execute the task.
```

```
        virtual uint32Run() override;

public:
        virtualbool Create(class FQueuedThreadPoolBase* InPool,uint32
        InStackSize,EThreadPriority        ThreadPriority);
        bool KillThread();
        void    DoWork(IQueuedWork* InQueuedWork);
};
```

- **TAsyncRunnable**:A task that runs asynchronously in a separate thread. It is a template class and is declared as follows:

```
// Engine\Source\Runtime\Core\Public\Async\Async.h

template<typename ResultType> class
TAsyncRunnable: public FRunnable {

public:
        virtual uint32Run() override;

private:
        TUniqueFunction<ResultType()>              Function;
        TPromise<ResultType> Promise;
        TFuture<FRunnableThread*> ThreadFuture;
};
```

- **FAsyncPurge**:Auxiliary class that provides destruction of UObject objects located in the working thread.

It can be seen from this that the FRunnable object cannot exist independently and always relies on threads to actually perform tasks.
In addition, it should be pointed out that FRenderingThread and FQueuedThread sound like real threads, but they are not. They are just executable objects used to handle certain specific tasks. In fact, they still rely on their internal FRunnableThread member objects for execution.

2.4.2.2 FRunnableThread

FRunnableThread is the parent class of runnable threads, providing a set of interfaces for managing the thread life cycle. The basic interfaces and analysis it provides are as follows:

```
// Engine\Source\Runtime\Core\Public\HAL\RunnableThread.h

classCORE_API FRunnableThread {

        staticuint32 RunnableTlsSlot;                        // FRunnableThreadofTLSThe slot index.

public:
        staticuint32GetTlsSlot();
        //Static class, used to create threads, needs to provide aFRunnableObject, used for tasks performed by threads.
        staticFRunnableThread*Create(FRunnable* InRunnable,constTCHAR* ThreadName, uint32 InStackSize =0,
```

```
                                                    EThreadPriority InThreadPri, uint64
InThreadAffinityMask,EThreadCreateFlags InCreateFlags);

    //Set the thread priority.
    virtualvoid SetThreadPriority( EThreadPriority NewPriority ); //Pause/resume
    running thread
    virtualvoid Suspend(boolbShouldPause =true);
    //Destroy a thread, usually requires specifying a wait markerbShouldWaitfortrue,Otherwise it may cause memory leaks or
    deadlocks! virtualbool Kill(boolbShouldWait =true); //Waiting for the execution to complete will block the calling thread.

    virtualvoid WaitForCompletion();

    constuint32GetThreadID()const; constFString&
    GetThreadName()const;

protected:
    FString ThreadName;
    FRunnable* Runnable;//Executed object
    FEvent* ThreadInitSyncEvent;//Thread initialization completion synchronization event to prevent the thread from executing tasks before
    initialization is complete. uint64 ThreadAffinityMask;//Affinity tags, used to specify thread preferencesCPUCore execution.
    TArray<FTlsAutoCleanup*> TlsInstances;//Need to be cleaned up together when the thread is consumedTlsObject. EThreadPriority
    ThreadPriority; uint32 ThreadID;


private:
    virtual     void   Tick();
};
```

It should be noted that FRunnableThread provides a static creation interface. When creating a thread, you need to specify an FRunnable object as the task to be executed by the thread. It is a basic parent class. The following are some core subclasses inherited from it and their analysis:

- **FRunnableThreadWin**:Thread implementation for Windows platform. Its interface and implementation are as follows:

```
// Engine\Source\Runtime\Core\Private\Windows\WindowsRunnableThread.h

classFRunnableThreadWin: public FRunnableThread {

    HANDLE Thread;//Thread handle

    //Thread callback interface, passed in as a parameter when creating a thread.
    static::DWORD STDCALL _ThreadProc( LPVOID pThis ) {

        check(pThis);
        return((FRunnableThreadWin*)pThis)->GuardedRun();
    }

    uint32    GuardedRun();
    uint32    Run();

public:
    //Conversion Priority
    static int TranslateThreadPriority(EThreadPriority Priority) {

        switch(Priority)
```

```
            {
            caseTPri_AboveNormal:returnTHREAD_PRIORITY_HIGHEST; caseTPri_Normal:
            returnTHREAD_PRIORITY_HIGHEST -1; caseTPri_BelowNormal:return
            THREAD_PRIORITY_HIGHEST -3; caseTPri_Highest:return
            THREAD_PRIORITY_HIGHEST; caseTPri_TimeCritical:return
            THREAD_PRIORITY_HIGHEST; caseTPri_Lowest:return
            THREAD_PRIORITY_HIGHEST -4;
            caseTPri_SlightlyBelowNormal:returnTHREAD_PRIORITY_HIGHEST -2; default
            :UE_LOG(LogHAL, Fatal, TEXT("Unknown Priority passed to TranslateThreadPriority()"));
returnTPri_Normal;
            }
      }

      //Setting Priorities
      virtualvoid SetThreadPriority( EThreadPriority NewPriority ) override {

            // Don't bother calling the OS if there is no need ThreadPriority =
            NewPriority; // Change the priority on the thread

            ::SetThreadPriority(Thread, TranslateThreadPriority(ThreadPriority));
      }

      virtualvoid Suspend(boolbShouldPause =true) override {

            check(Thread);
            if(bShouldPause ==true) {

                  SuspendThread(Thread);
            }
            else
            {
                  ResumeThread(Thread);
            }
      }

      virtualbool Kill(boolbShouldWait =false) override {

            check(Thread &&"Did you forget to call Create()?"); boolbDidExitOK =
            true; //
                  Stop firstRunnableobject, giving it a chance to clean up the
            if    data (Runnable)
            {
                  Runnable->Stop();
            }
            // Wait for the thread to complete processing.
            if(bShouldWait ==true) {

                  // Wait indefinitely for the thread to finish.                  IMPORTANT:    It's not
safe to just go and
                  // kill the thread with TerminateThread() as it could have a mutex lock shared
that's
                  // with a thread that's continuing to run, which would cause that other
thread    to
                  // dead-lock. (This can manifest itself in code as simple as the
synchronization
                  // object that is used by our logging output classes. Trust us, we've
seen it!)
                  WaitForSingleObject(Thread,INFINITE);
```

```
            }
            //Close the thread handle
            CloseHandle(Thread);
            Thread =NULL;


            returnbDidExitOK;
        }

        virtualvoid WaitForCompletion( ) override {

            // Block until this thread exits
            WaitForSingleObject(Thread,INFINITE);
        }

protected:

        virtualbool CreateInternal( FRunnable* InRunnable,constTCHAR* InThreadName,
            uint32 InStackSize =0,
            EThreadPriority InThreadPri = TPri_Normal, uint64 InThreadAffinityMask =0, EThreadCreateFlags
            InCreateFlags = EThreadCreateFlags::None) override
        {
            static boolbOnce =false; if(!bOnce)

            {
                bOnce =true;
                ::SetThreadPriority(::GetCurrentThread(),
TranslateThreadPriority(TPri_Normal));// set the main thread to be normal, since this is no longer the windows
default.
            }

            check(InRunnable);
            Runnable = InRunnable;
            ThreadAffinityMask = InThreadAffinityMask;

             //Creates an initialization complete synchronization event.
            ThreadInitSyncEvent                = FPlatformProcess::GetSynchEventFromPool(true);

            ThreadName = InThreadName ? InThreadName : TEXT("Unnamed UE4");

            // Create the new thread {

                LLM_SCOPE(ELLMTag::ThreadStack);
                LLM_PLATFORM_SCOPE(ELLMTag::ThreadStackPlatform);
                // add in the thread size, since it's allocated in a black box we can't
track
                LLM(FLowLevelMemTracker::Get().OnLowLevelAlloc(ELLMTracker::Default, InStackSize));
nullptr,
                LLM(FLowLevelMemTracker::Get().OnLowLevelAlloc(ELLMTracker::Platform, InStackSize));
nullptr,

                //CallWindows APICreate a thread.
                Thread = CreateThread(NULL, InStackSize, _ThreadProc, this,
STACK_SIZE_PARAM_IS_A_RESERVATION | CREATE_SUSPENDED, (::DWORD *)&ThreadID);
            }

            // If it fails, clear all the vars if(Thread ==NULL)
            {
```

```
            Runnable = nullptr;
        }
        else
        {
            //Added to the thread manager.
            FThreadManager::Get().AddThread(ThreadID,                this);
            ResumeThread(Thread);

            // Let the thread start up ThreadInitSyncEvent-
            >Wait(INFINITE);

            SetThreadPriority(InThreadPri);
        }
        //Clean up synchronization events
        FPlatformProcess::ReturnSynchEventToPool(ThreadInitSyncEvent);
        ThreadInitSyncEvent = nullptr;
        returnThread !=NULL;
    }
};
```

As can be seen from the above code, the threads of the Windows platform directly call the Windows API to create and synchronize information, thereby realizing the platform abstraction of threads and extracting them from platform dependencies.

- **FRunnableThreadPThread**:The parent class of POSIX Thread (PThread for short), commonly used in Unix-like POSIX systems, such as Linux, Solaris, Apple, etc. Its implementation is similar to that of the Windows platform, so its code analysis will not be expanded here. Its subclasses are:
    - **FRunnableThreadApple**:Thread of Apple system (MacOS, iOS).
    - **FRunnableThreadAndroid**:Android system thread.
    - **FRunnableThreadUnix**:Unix system thread.
- **FRunnableThreadHoloLens**:HoloLens system thread.
- **FFakeThread**:A fake thread, a substitute when multithreading is disabled, actually running on a single thread.

FRunnable and FRunnableThread complement each other and are indispensable. One is the carrier of the operation, and the other is the content of the operation. The following is an application example of them:

```
//DerivationFRunnable
classFMyRunnable: public FRunnable {

    boolbStop;
public:
    virtualbool Init(void) {

        bStop =false;
```

```
        returntrue;
    }

    virtual uint32Run(void) {

        for(int32 i =0; i <10&& !bStop; i++) {

            FPlatformProcess::Sleep(1.0f);
        }

        return0;
    }

    virtualvoid Stop(void) {

        bStop =true;
    }

    virtualvoid Exit(void) {

    }

};

void TestRunnableAndRunnableThread() {

    //createRunnableObject
    FMyRunnable* MyRunnable = new FMyRunnable; //
    Create a thread, pass inMyRunnable
    FRunnableThread* MyThread = FRunnableThread::Create(MyRunnable, TEXT("MyRunnable"));

    //Pause the current thread
    FPlatformProcess::Sleep(4.0f);

    //Waiting for a thread to finish
    MyRunnable->Stop();
    MyThread->WaitForCompletion();

    //Clean the data.
    delete    MyThread;
    delete    MyRunnable;
}
```

Careful students should have noticed that when creating a thread, the thread will be added to FThreadManager, which means that all threads are managed by FThreadManager. The following is the declaration of FThreadManager:

```
// Engine\Source\Runtime\Core\Public\HAL\ThreadManager.h

classFThreadManager
{
    FCriticalSection ThreadsCritical;//Modify the thread listThreadsCritical area static
    boolbIsInitialized;

    TMap<uint32,classFRunnableThread*,TInlineSetAllocator<256>>Threads;//Thread list, note that the data structure
isMap, KeyIt is a threadID.
```

```
public:
    void AddThread(uint32 ThreadId, class FRunnableThread* Thread);//Adding threads void
    RemoveThread(class FRunnableThread* Thread);//Deleting a Thread

    void Tick();//Frame update, only forFFakeThreadkick in.

    constFString&GetThreadName(uint32 ThreadId);
    void ForEachThread(TFunction<void(uint32, class FRunnableThread*)> Func);//Traversing Threads

    static    bool IsInitialized();
    static    FThreadManager&Get();
};
```

2.4.2.3 QueuedWork

This section will explain the UE's queued QueuedWork system, including IQueuedWork, TAsyncQueuedWork, FQueuedThreadPool, FQueuedThreadPoolBase, etc.

- **IQueuedWork and TAsyncQueuedWork**

IQueuedWork is a set of abstract interfaces that stores a set of queued task objects and will be executed by the FQueuedThreadPool thread pool object. The interface of IQueuedWork is as follows:

```
// Engine\Source\Runtime\Core\Public\Misc\IQueuedWork.h

classIQueuedWork
{
public:
    virtualvoid DoThreadedWork() =0;//Execute queued tasks.
    virtualvoid Abandon() =0;//Give up execution early and notify all objects in the queue to clean up data.
};
```

Since IQueuedWork is just an abstract class and does not actually execute code, the main subclass TAsyncQueuedWork is responsible for implementing the code. The following is the declaration and implementation of TAsyncQueuedWork:

```
// Engine\Source\Runtime\Core\Public\Async\Async.h

template<typename ResultType>
classTAsyncQueuedWork: public IQueuedWork {

public:
    virtualvoid DoThreadedWork() override {

        SetPromise(Promise, Function); delete
        this;
    }

    virtualvoid Abandon() override {

        // not supported
    }
```

```
private:
    TUniqueFunction<ResultType()> Function;//List of functions to be executed.
    TPromise<ResultType> Promise;//Objects used for synchronization
};
```

- **FQueuedThreadPool and FQueuedThreadPoolBase**

Similar to FRunnable and FRunnableThread, TAsyncQueuedWork cannot execute tasks independently and needs to rely on FQueuedThreadPool to execute. The following is the declaration of FQueuedThreadPool:

```
// Engine\Source\Runtime\Core\Public\Misc\QueuedThreadPool.h

//implementIQueuedWorkThread pool for task
lists. classFQueuedThreadPool {

public:
    //Creates the specified number of threads, with the specified stack size and priority.
    virtualbool Create( uint32 InNumQueuedThreads, uint32 StackSize = (32*1024), EThreadPriority
ThreadPriority=TPri_Normal ) =0;
    //Destroys a background thread within a thread.
    virtualvoid Destroy() =0;
    //Add a task to the queue. If there is an available thread, it will be executed immediately; otherwise it will
    be executed later. virtualvoid AddQueuedWork(IQueuedWork* InQueuedWork) =0; //Cancel the specified
    queued task.
    virtualbool RetractQueuedWork(IQueuedWork* InQueuedWork) =0; //Get the number
    of threads.
    virtual int32GetNumThreads()const=0;

public:
    //Create a thread pool object.
    staticFQueuedThreadPool* //          Allocate();
    Override the stack size.
    staticuint32 OverrideStackSize;
};
```

As can be seen above, FQueuedThreadPool is an abstract class that only provides an interface but no implementation. In fact, the implementation is in FQueuedThreadPoolBase, as follows:

```
// Engine\Source\Runtime\Core\Private\HAL\ThreadingBase.cpp

classFQueuedThreadPoolBase: public FQueuedThreadPool {

protected:
    TArray<IQueuedWork*> QueuedWork;//List of tasks that need to be performed
    TArray<FQueuedThread*> QueuedThreads;//Available threads in the thread pool
    TArray<FQueuedThread*> AllThreads;                    //All threads in the thread
    pool FCriticalSection* SynchQueue;//Synchronous critical section boolTimeToDie;//
    Timeout Marker

public:
    FQueuedThreadPoolBase()
        : SynchQueue(nullptr)
        , TimeToDie(0)
```

```
    { }
    virtual ~FQueuedThreadPoolBase() {

        Destroy();
    }

    virtualbool Create(uint32 InNumQueuedThreads,uint32 StackSize = (32* 1024),EThreadPriority
ThreadPriority=TPri_Normal) override
    {
        //Handle synchronization locks.
        boolbWasSuccessful =true; check(SynchQueue
        == nullptr); SynchQueue = new FCriticalSection();
        FScopeLockLock(SynchQueue);

        // Presize the array so there is no extra memory allocated
        check(QueuedThreads.Num() ==0);
        QueuedThreads.Empty(InNumQueuedThreads);

        if(OverrideStackSize > StackSize ) {

            StackSize = OverrideStackSize;
        }

         //Create a thread, note that it is createdFQueuedThread.
        for(uint32 Count =0; Count < InNumQueuedThreads && bWasSuccessful ==true;
Count++)
        {
            FQueuedThread* pThread = new FQueuedThread(); //use
            FQueuedThreadObject creates the actual thread.
            if(pThread->Create(this,StackSize,ThreadPriority) ==true) {

                QueuedThreads.Add(pThread);
                AllThreads.Add(pThread);
            }
            else
            {
                //Creation failed, clean up the thread object.
                bWasSuccessful =false; delete
                pThread;
            }
        }
        //Failed to create thread pool, cleaning up data.
        if(bWasSuccessful ==false) {


            Destroy();
        }
        returnbWasSuccessful;
    }

    virtualvoid Destroy() override {

        if(SynchQueue)
        {
            {
                FScopeLock     Lock(SynchQueue);
                TimeToDie = 1;
                FPlatformMisc::MemoryBarrier(); // Clean
                up all queued objects
```

```
                for(int32 Index =0; Index < QueuedWork.Num(); Index++) {

                        QueuedWork[Index]->Abandon();
                }
                // Empty out the invalid pointers
                QueuedWork.Empty();
        }
        //Wait for all threads to complete execution. Note that synchronization time is not used here, but a mechanism similar to a spin lock is
        used. while (1)
        {
                {
                        //accessAllThreadsandQueuedThreadsLock the critical section first when accessing data. Prevent other threads from modifying
                        the data. FScopeLockLock(SynchQueue);
                        if(AllThreads.Num() == QueuedThreads.Num()) {

                                break;
                        }
                }
                FPlatformProcess::Sleep(0.0f);//Switch the current thread time slice to prevent the current thread from occupyingcpuclock.
        }
        //    Delete all threads.
        {
                FScopeLockLock(SynchQueue);
                // Now tell each thread to die and delete those for(int32 Index =0; Index <
                AllThreads.Num(); Index++) {

                        AllThreads[Index]->KillThread(); delete
                        AllThreads[Index];
                }
                QueuedThreads.Empty();
                AllThreads.Empty();
        }
        //    Remove the synchronization lock.
        delete     SynchQueue;
        SynchQueue    = nullptr;
    }
}

int32GetNumQueuedJobs()const {

    returnQueuedWork.Num();
}
virtual int32GetNumThreads()const {

    returnAllThreads.Num();
}

//Add a queued task.
void AddQueuedWork(IQueuedWork* InQueuedWork) override {

    check(InQueuedWork != nullptr);

    if(TimeToDie)
    {
        InQueuedWork->Abandon();
        return;
    }
```

```
        check(SynchQueue);

        FQueuedThread* Thread = nullptr;

        {
            //The critical section needs to be locked before operating all data in the
            thread pool. FScopeLocksl(SynchQueue);
            constint32 AvailableThreadCount = QueuedThreads.Num();

            //There are no available threads. Add the task to the task queue
            and execute it later. if(AvailableThreadCount ==0) {

                QueuedWork.Add(InQueuedWork);
                return;
            }

            //Gets a thread from the available thread pool and removes it from the
            available thread pool. constint32 ThreadIndex = AvailableThreadCount -1;

            Thread = QueuedThreads[ThreadIndex]; QueuedThreads.RemoveAt(ThreadIndex,1,/* do not allow
            shrinking */false);
        }

        //Execute the task
        Thread->DoWork(InQueuedWork);
    }

    virtualbool RetractQueuedWork(IQueuedWork* InQueuedWork) override {

        if(TimeToDie)
        {
            returnfalse;// no special consideration for this, refuse the retraction and shutdown proceed
let
        }
        check(InQueuedWork != nullptr);
        check(SynchQueue);
        FScopeLocksl(SynchQueue);
        return!!QueuedWork.RemoveSingle(InQueuedWork);
    }

    //If there is an available task, get one and execute it, otherwise return the thread to the available thread pool. This interface is provided
    byFQueuedThreadCall. IQueuedWork*ReturnToPoolOrGetNextJob(FQueuedThread* InQueuedThread) {

        check(InQueuedThread != nullptr);
        IQueuedWork* Work = nullptr;
        // Check to see if there is any work to be done FScopeLocksl
        (SynchQueue); if(TimeToDie)

        {
            check(!QueuedWork.Num());// we better not have anything if we are dying
        }
        if(QueuedWork.Num() >0) {

            // Grab the oldest work in the queue. This is slower than // getting the most
            recent but prevents work from being // queued and never done

            Work = QueuedWork[0];
            // Remove it from the list so no one else grabs it
```

```
                QueuedWork.RemoveAt(0,1,/* do not allow shrinking */false);
            }
            if  (!Work)
            {
                // There was no work to be done, so add the thread to the pool
                QueuedThreads.Add(InQueuedThread);
            }
            return  Work;
        }
};
```

The ReturnToPoolOrGetNextJob interface above is not called by FQueuedThreadPoolBase, but is actively called by the FQueuedThread object that is executing a task and has completed it, as shown below:

```
uint32FQueuedThread::Run() {

    while(!TimeToDie.Load(EMemoryOrder::Relaxed)) {

        boolbContinueWaiting =true;

        (.....)

        //  Make the event wait.
        if  (bContinueWaiting)
        {
            DoWorkEvent->Wait();
        }

        IQueuedWork* LocalQueuedWork = QueuedWork;
        QueuedWork = nullptr;
        FPlatformMisc::MemoryBarrier();
        check(LocalQueuedWork || TimeToDie.Load(EMemoryOrder::Relaxed));// well you woke me up, where is the
job or termination request?
            //Continuously obtain tasks from the thread pool and execute them,  Until all tasks in the thread pool are completed.
        while  (LocalQueuedWork)
        {
            //  Execute the task.
            LocalQueuedWork->DoThreadedWork(); //Get the
            next task from the thread pool.
            LocalQueuedWork = OwningThreadPool->ReturnToPoolOrGetNextJob(this);
        }
    }
    return0;
}
```

As can be seen from the above, the data and interfaces of FQueuedThreadPool and FQueuedThread are cleverly coordinated to execute tasks in parallel.

- **GThreadPool**

The mechanism of the thread pool has been described. Let's talk about the initialization process of UE's global thread pool GThreadPool. This process is in FEngineLoop::PreInitPreStartupScreen.1.4.6.1 Engine pre-initialization has been mentioned:

```
// Engine\Source\Runtime\Launch\Private\LaunchEngineLoop.cpp

int32FEngineLoop::PreInitPreStartupScreen(constTCHAR* CmdLine) {

    (......)

    {
        TRACE_THREAD_GROUP_SCOPE("ThreadPool"); //Creating a
        global thread pool
        GThreadPool = FQueuedThreadPool::Allocate();
        int32 NumThreadsInThreadPool = FPlatformMisc::NumberOfWorkerThreadsToSpawn();

        //  If it is pure server mode, the thread pool has only one thread.
        if  (FPlatformProperties::IsServerOnly())
        {
            NumThreadsInThreadPool =1;
        }
        //  Create worker threads equal to the number of threads.
        verify(GThreadPool->Create(NumThreadsInThreadPool, StackSize *1024,
        TPri_SlightlyBelowNormal));
    }

    (......)
}
```

If we need GThreadPool to do something for us, the usage example is as follows:

```
// Engine\Source\Runtime\Engine\Private\ShadowMap.cpp

//  Multithreaded texture encoding
if  (bMultithreadedEncode)
{
    //Completed tasks counter.
    FThreadSafeCounterCounter(PendingTextures.Num()); //List of
    texture tasks to be encoded
    TArray<FAsyncEncode<FShadowMapPendingTexture>> AsyncEncodeTasks;
    AsyncEncodeTasks.Empty(PendingTextures.Num());
    //Create all tasks and add them toAsyncEncodeTasksList. for
    (auto& PendingTexture : PendingTextures) {

        PendingTexture.CreateUObjects(); //create
        AsyncEncodeTask
        autoAsyncEncodeTask = new
(AsyncEncodeTasks)FAsyncEncode<FShadowMapPendingTexture>(&PendingTexture,
LightingScenario, Counter, TextureCompressorModule);
        //WillAsyncEncodeTaskJoin the global thread pool and execute.
        GThreadPool->AddQueuedWork(AsyncEncodeTask);
    }
    //If there are still unfinished tasks, put the current thread into
    sleep state. while(Counter.GetValue() >0) {

        GWarn->UpdateProgress(Counter.GetValue(), PendingTextures.Num());
        FPlatformProcess::Sleep(0.0001f);
    }
}
```

2.4.2.4 TaskGraph

TaskGraphliterally means task graph, and the graph used is**DAG**(Directed Acyclic Graph). You can specify dependencies, specify predecessor and successor tasks, but there cannot be cyclic dependencies. It is the most complex parallel task system in UE so far, and the complexity of the concepts and operating mechanisms involved has increased dramatically. This section will spend a lot of space to describe them, aiming to explain their mechanisms and principles clearly.

- **FBaseGraphTask**

FBaseGraphTask is a task running on TaskGraph. It is a base parent class, and its derived specific task subclasses will execute tasks. Its declaration (excerpt) is as follows:

```
// Engine\Source\Runtime\Core\Public\Async\TaskGraphInterfaces.h

classFBaseGraphTask
{
protected:
	FBaseGraphTask(int32	InNumberOfPrerequistitesOutstanding);

	//The prerequisite task is completed or partially completed.
	void PrerequisitesComplete(ENamedThreads::Type CurrentThread, int32
NumAlreadyFinishedPrequistes,boolbUnlock =true);

	//Execute tasks with conditions (predecessors have been completed)
	void ConditionalQueueTask(ENamedThreads::Type CurrentThread) {

		if(NumberOfPrerequistitesOutstanding.Decrement()==0) {

			QueueTask(CurrentThread);
		}
	}

private:
	//The actual execution of the task is implemented by the subclass.
	virtualvoid ExecuteTask(TArray<FBaseGraphTask*>& NewTasks, ENamedThreads::Type CurrentThread)=0;


	//JoinTaskGraphin the task queue.
	void QueueTask(ENamedThreads::Type CurrentThreadIfKnown) {

		checkThreadGraph(LifeStage.Increment() ==		int32(LS_Queued));
		FTaskGraphInterface::Get().QueueTask(this,		ThreadToExecuteOn,
		CurrentThreadIfKnown);
	}

	ENamedThreads::Type ThreadToExecuteOn;//The type of thread that executes the
	task FThreadSafeCounter NumberOfPrerequistitesOutstanding;//		Counter before executing the task
};
```

- **TGraphTask**

The only subclass of FBaseGraphTask, TGraphTask, inherits the code to complete the task execution. The declaration and implementation of TGraphTask are as follows:

```
template<typename TTask>
classTGraphTaskfinal: public FBaseGraphTask {

public:
    //Auxiliary class for constructing tasks.
    classFConstructor
    {
    public:
        //createTTaskTask object, then setTGraphTaskThe task data is then executed at the appropriate
        time. template<typename...T>
        FGraphEventRefConstructAndDispatchWhenReady(T&&... Args) {

            new ((void*)&Owner->TaskStorage) TTask(Forward<T>(Args)...); returnOwner-
            >Setup(Prerequisites, CurrentThreadIfKnown);
        }

        //createTTaskTask object, then setTGraphTaskThe task's data is held but not executed.
        template<typename...T>
        TGraphTask*ConstructAndHold(T&&... Args) {

            new ((void*)&Owner->TaskStorage) TTask(Forward<T>(Args)...); returnOwner-
            >Hold(Prerequisites, CurrentThreadIfKnown);
        }

    private:
        TGraphTask*                 Owner;//LocatedTGraphTaskObject.
        constFGraphEventArray*      Prerequisites;//Prerequisite tasks.
        ENamedThreads::Type         CurrentThreadIfKnown;
    };

    //Create a task, note that what is returned isFConstructorobject to perform subsequent operations on the task.
    staticFConstructorCreateTask(constFGraphEventArray* Prerequisites =NULL, ENamedThreads::Type
CurrentThreadIfKnown = ENamedThreads::AnyThread)
    {
        int32 NumPrereq = Prerequisites ? Prerequisites->Num() :0; if(sizeof
        (TGraphTask) <= FBaseGraphTask::SMALL_TASK_SIZE) {

            void*Mem = FBaseGraphTask::GetSmallTaskAllocator().Allocate(); returnFConstructor(new (Mem)
            TGraphTask(TTask::GetSubsequentsMode() ==
ESubsequentsMode::FireAndForget?NULL: FGraphEvent::CreateGraphEvent(), NumPrereq), Prerequisites,
CurrentThreadIfKnown);
        }
        returnFConstructor(new TGraphTask(TTask::GetSubsequentsMode() ==
ESubsequentsMode::FireAndForget ?NULL: FGraphEvent::CreateGraphEvent(), NumPrereq), Prerequisites,
CurrentThreadIfKnown);
    }

    void Unlock(ENamedThreads::Type CurrentThreadIfKnown = ENamedThreads::AnyThread) {

        ConditionalQueueTask(CurrentThreadIfKnown);
    }

    FGraphEventRefGetCompletionEvent() {

        returnSubsequents;
    }
```

```
private:
	//Execute the task
	void ExecuteTask(TArray<FBaseGraphTask*>& NewTasks, ENamedThreads::Type CurrentThread) override

	{
		(.....)

		//Handle subsequent tasks.
		if(TTask::GetSubsequentsMode() == ESubsequentsMode::TrackSubsequents) {

			Subsequents->CheckDontCompleteUntilIsEmpty();// we can only add wait for
tasks while executing the task
		}

		//Execute the task
		TTask& Task = *(TTask*)&TaskStorage; {

			FScopeCycleCounterScope(Task.GetStatId(),		true);
			Task.DoTask(CurrentThread, Subsequents);
			Task.~TTask();
			checkThreadGraph(ENamedThreads::GetThreadIndex(CurrentThread)		<=
ENamedThreads::GetRenderThread() || FMemStack::Get().IsEmpty());// you must mark and pop memstacks if you use
them in tasks! Named threads are excepted.
		}

		TaskConstructed =false;

		//Execute subsequent tasks.
		if(TTask::GetSubsequentsMode() == ESubsequentsMode::TrackSubsequents) {

			FPlatformMisc::MemoryBarrier(); Subsequents-
			>DispatchSubsequents(NewTasks, CurrentThread);
		}

		//Release the task object data.
		if(sizeof(TGraphTask) <= FBaseGraphTask::SMALL_TASK_SIZE) {

			this->TGraphTask::~TGraphTask();
			FBaseGraphTask::GetSmallTaskAllocator().Free(this);
		}
		else
		{
			delete this;
		}
	}

	//Set up prerequisite tasks.
	void SetupPrereqs(constFGraphEventArray* Prerequisites, ENamedThreads::Type
CurrentThreadIfKnown,boolbUnlock)
	{
		checkThreadGraph(!TaskConstructed);
		TaskConstructed =true;
		TTask& Task = *(TTask*)&TaskStorage;
		SetThreadToExecuteOn(Task.GetDesiredThread()); int32
		AlreadyCompletedPrerequisites =0; if(Prerequisites)

		{
			for(int32 Index =0; Index < Prerequisites->Num(); Index++)
```

```
                {
                    check((*Prerequisites)[Index]);
                    if(!(*Prerequisites)[Index]->AddSubsequent(this)) {

                        AlreadyCompletedPrerequisites++;
                    }
                }
            }
            PrerequisitesComplete(CurrentThreadIfKnown, AlreadyCompletedPrerequisites, bUnlock);

        }

        //Set task data.
        FGraphEventRefSetup(constFGraphEventArray* Prerequisites =NULL, ENamedThreads::Type CurrentThreadIfKnown
= ENamedThreads::AnyThread)
        {
            FGraphEventRef ReturnedEventRef = Subsequents;// very important so that this doesn't get destroyed
before we return
            SetupPrereqs(Prerequisites, CurrentThreadIfKnown,true); return
            ReturnedEventRef;
        }

        //Holds task data.
        TGraphTask*Hold(constFGraphEventArray* Prerequisites =NULL, ENamedThreads::Type
CurrentThreadIfKnown = ENamedThreads::AnyThread)
        {
            SetupPrereqs(Prerequisites, CurrentThreadIfKnown,false); returnthis;

        }

        //Create a task.
        staticFConstructorCreateTask(FGraphEventRef SubsequentsToAssume,const FGraphEventArray*
Prerequisites =NULL, ENamedThreads::Type CurrentThreadIfKnown = ENamedThreads::AnyThread)

        {
            if(sizeof(TGraphTask) <= FBaseGraphTask::SMALL_TASK_SIZE) {

                void*Mem = FBaseGraphTask::GetSmallTaskAllocator().Allocate();
                returnFConstructor(new (Mem) TGraphTask(SubsequentsToAssume, Prerequisites?
Prerequisites->Num() :0), Prerequisites, CurrentThreadIfKnown);
            }
            returnFConstructor(new TGraphTask(SubsequentsToAssume, Prerequisites ? Prerequisites-
>Num() :0), Prerequisites, CurrentThreadIfKnown);
        }

        TAlignedBytes<sizeof(TTask),alignof(TTask)> TaskStorage;//The task object to be executed. bool
                                                TaskConstructed;
        FGraphEventRef                          Subsequents;//Subsequent task synchronization object.
};
```

- **TAsyncGraphTask**

As can be seen above, although TGraphTask is a task, the actual task it performs is the template class of TTask. The basic form of TTask is given in the UE comment:

```
classFGenericTask
{
	TSomeType	SomeArgument;
public:
	FGenericTask(TSomeType InSomeArgument)//References cannot be used, pointers can be used instead.
		: SomeArgument(InSomeArgument)
	{
		// Usually the constructor doesn't do anything except save the arguments for use in DoWork or
		GetDesiredThread.
	}
	~FGenericTask()
	{
		// you will be destroyed immediately after you execute. Might as well do cleanup in DoWork, but you could
also use a destructor.
	}
	FORCEINLINE TStatIdGetStatId()const {

		RETURN_QUICK_DECLARE_CYCLE_STAT(FGenericTask, STATGROUP_TaskGraphTasks);
	}

	[static] ENamedThreads::TypeGetDesiredThread() {

		returnENamedThreads::[named thread or AnyThread];
	}
	void DoTask(ENamedThreads::Type CurrentThread,constFGraphEventRef&
	MyCompletionGraphEvent)
	{
		// The arguments are useful for setting up other tasks. // Do work here,
		probably using SomeArgument. MyCompletionGraphEvent -

> DontCompleteUntil(TGraphTask<FSomeChildTask>::CreateTask(NULL,CurrentThread).ConstructAnd
DispatchWhenReady());
	}
};
```

However, if we need to customize our own tasks, we can directly use or derive the TAsyncGraphTask class without starting from scratch. TAsyncGraphTask and its parent class FAsyncGraphTaskBase are declared as follows:

```
// Engine\Source\Runtime\Core\Public\Async\Async.h

//Post-task mode
namespace ESubsequentsMode
{
	enum Type
	{
		TrackSubsequents,		// Tracking subsequent tasks
		FireAndForget	//No need to track task dependencies, thread synchronization can be avoided, and execution efficiency can be improved.
	};
}

classFAsyncGraphTaskBase {

public:
	TStatIdGetStatId()const
```

```
    {
        return     GET_STATID(STAT_TaskGraph_OtherTasks);
    }

    //Task post-order mode.
    staticESubsequentsMode::TypeGetSubsequentsMode() {

        returnESubsequentsMode::FireAndForget;
    }
};

template<typename ResultType>
classTAsyncGraphTask: public FAsyncGraphTaskBase {

public:
    //Structuring tasks,InFunctionThis is the code segment that needs to be
    executed. TAsyncGraphTask(TUniqueFunction<ResultType()>&&                InFunction,        TPromise<ResultType>&&
InPromise, ENamedThreads::Type InDesiredThread = ENamedThreads::AnyThread)
        : Function(MoveTemp(InFunction)) ,
        Promise(MoveTemp(InPromise))
        , DesiredThread(InDesiredThread)
    { }

public:
    //Execute the task
    void DoTask(ENamedThreads::Type CurrentThread,constFGraphEventRef&
    MyCompletionGraphEvent)
    {
        SetPromise(Promise, Function);
    }

    ENamedThreads::TypeGetDesiredThread() {

        returnDesiredThread;
    }

    TFuture<ResultType>GetFuture() {

        returnPromise.GetFuture();
    }

private:
    TUniqueFunction<ResultType()> Function;//The function object to be executed.
    TPromise<ResultType> Promise;//Synchronization object. ENamedThreads::Type
    DesiredThread;//The type of thread to execute.
};
```

- **FTaskThreadBase**

FTaskThreadBase is the thread parent class for executing tasks. It defines a set of interfaces for setting and operating tasks, as follows:

```
classFTaskThreadBase: public FRunnable, FSingleThreadRunnable {

public:
    FTaskThreadBase()
```

```
        : ThreadId(ENamedThreads::AnyThread) ,
        PerThreadIDTLSSlot(0xffffffff)
        , OwnerWorker(nullptr)
    {
        NewTasks.Reset(128);
    }

    //Set up the data.
    void Setup(ENamedThreads::Type InThreadId, uint32 InPerThreadIDTLSSlot, FWorkerThread* InOwnerWorker)

    {
        ThreadId = InThreadId;
        check(ThreadId >=       0);
        PerThreadIDTLSSlot       = InPerThreadIDTLSSlot;
        OwnerWorker = InOwnerWorker;
    }

    //Initialize from the current thread.
    voidInitializeForCurrentThread() {

        //Set platform-specificTLS.
        FPlatformTLS::SetTlsValue(PerThreadIDTLSSlot,                OwnerWorker);
    }

    ENamedThreads::TypeGetThreadId()const;

    //Used for named threads to process tasks until the thread is idle orRequestQuitis
    called. virtualvoid ProcessTasksUntilQuit(int32 QueueIndex) =0;

    //Used for named threads to process tasks until the thread is idle or
    RequestQuitis called. virtual uint64ProcessTasksUntilIdle(int32 QueueIndex);

    //Request to exit. Will cause the thread to exit to the caller when it is idle. If it is a named thread,ProcessTasksUntilQuitIt is used to return to the caller;
the unnamed thread is closed directly.
    virtualvoid RequestQuit(int32 QueueIndex) =0;

    //Enqueue task, assumingthisThe thread is the same as the current thread. If it is a named thread, it will go directly
    to the private queue. virtualvoid EnqueueFromThisThread(int32 QueueIndex, FBaseGraphTask* Task);

    //Enqueue task, assumingthisthread is not the same as the current thread.
    virtualbool EnqueueFromOtherThread(int32 QueueIndex, FBaseGraphTask* Task);

    //Wake up the thread.
    virtualvoid WakeUp();

    //Check whether the task is being processed.
    virtualbool IsProcessingTasks(int32 QueueIndex) =0;

    //Single thread frame update
    virtualvoid Tick() override {

        ProcessTasksUntilIdle(0);
    }

    // FRunnable API

    virtualbool Init() override {
```

```
        InitializeForCurrentThread(); returntrue
        ;
    }
    virtual uint32Run() override {

        check(OwnerWorker);// make sure we are started up
        ProcessTasksUntilQuit(0);
        FMemory::ClearAndDisableTLSCachesOnCurrentThread(); return0;

    }
    virtualvoid Stop() override {

        RequestQuit(-1);
    }
    virtualvoid Exit() override {

    }
    virtualFSingleThreadRunnable*GetSingleThreadInterface() override {

        returnthis;
    }

protected:
    ENamedThreads::Type                ThreadId;//Threadsid(Thread Index)
    uint32                             PerThreadIDTLSSlot;//TLSgroove. IsStalled;//Blocked
    FThreadSafeCounter                 counter. Used to trigger blocked signal.
    TArray<FBaseGraphTask*> NewTasks;//A list of pending tasks.
    FWorkerThread* OwnerWorker;//The worker thread object.
};
```

FTaskThreadBase is just an abstract class, and the specific implementation is completed by the subclasses FNamedTaskThread and FTaskThreadAnyThread.

Among them,**FNamedTaskThread**handles the tasks of named threads:

```
//A named task thread.
classFNamedTaskThread: public FTaskThreadBase {

public:
    //Used for named threads to process tasks until the thread is idle orRequestQuitis
    called. virtualvoid ProcessTasksUntilQuit(int32 QueueIndex) override {

        check(Queue(QueueIndex).StallRestartEvent);// make sure we are started up

        Queue(QueueIndex).QuitForReturn =false; verify(+
        +Queue(QueueIndex).RecursionGuard ==1);

        //  Continuously loops to process queued tasks until exited, closed, or the platform does not support multithreading.
        do
        {
            ProcessTasksNamedThread(QueueIndex,
FPlatformProcess::SupportsMultithreading());
        }while(!Queue(QueueIndex).QuitForReturn && !Queue(QueueIndex).QuitForShutdown &&
  FPlatformProcess::SupportsMultithreading());// @Hack - quit now when running with only one thread.

        verify(!--Queue(QueueIndex).RecursionGuard);
```

```
}

//Used for named threads to process tasks until the thread is idle orRequestQuitis called.
virtual uint64ProcessTasksUntilIdle(int32 QueueIndex) override {

	check(Queue(QueueIndex).StallRestartEvent);// make sure we are started up

	Queue(QueueIndex).QuitForReturn =false; verify(+
	+Queue(QueueIndex).RecursionGuard ==1);
	uint64 ProcessedTasks = ProcessTasksNamedThread(QueueIndex,false); verify(!--
	Queue(QueueIndex).RecursionGuard);
	returnProcessedTasks;
}

//Processing tasks.
uint64ProcessTasksNamedThread(int32 QueueIndex,boolbAllowStall) {

	uint64 ProcessedTasks =0;

	(.....)

	TStatId StallStatId;
	boolbCountAsStall =false;

	(.....)

	while	(!Queue(QueueIndex).QuitForReturn)
	{
		//	Get the task from the head of the queue.
		FBaseGraphTask* Task = Queue(QueueIndex).StallQueue.Pop(0, bAllowStall);
		TestRandomizedThreads();
		if(!Task)
		{
			if(bAllowStall)
			{
				{
					FScopeCycleCounterScope(StallStatId);
					Queue(QueueIndex).StallRestartEvent->Wait(MAX_uint32,
bCountAsStall);
					if(Queue(QueueIndex).QuitForShutdown) {

						returnProcessedTasks;
					}
					TestRandomizedThreads();
				}
				continue;
			}
			else
			{
				break;// we were asked to quit
			}
		}
		else//Task is not empty
		{
			//Execute the task.
			Task->Execute(NewTasks, ENamedThreads::Type(ThreadId | (QueueIndex <<
ENamedThreads::QueueIndexShift)));
			ProcessedTasks++;
```

```
                TestRandomizedThreads();
            }
        }
        returnProcessedTasks;
    }

    virtualvoid EnqueueFromThisThread(int32 QueueIndex, FBaseGraphTask* Task) override {

        checkThreadGraph(Task && Queue(QueueIndex).StallRestartEvent);// make sure we are started up

        uint32 PriIndex = ENamedThreads::GetTaskPriority(Task->ThreadToExecuteOn)?0:1; int32 ThreadToStart =
        Queue(QueueIndex).StallQueue.Push(Task, PriIndex); check(ThreadToStart <0);// if I am stalled, then how can I
        be queuing a task?
    }

    virtualvoid RequestQuit(int32 QueueIndex) override {

        // this will not work under arbitrary circumstances. For example you should not attempt to stop threads
unless they are known to be idle.
        if(!Queue(0).StallRestartEvent) {

            return;
        }
        if(QueueIndex ==-1) {

            // we are shutting down
            checkThreadGraph(Queue(0).StallRestartEvent);// make sure we are started up
            checkThreadGraph(Queue(1).StallRestartEvent);// make sure we are started up Queue(0
            ).QuitForShutdownQueue(1).QuitF=orStrhuuet;down
                                                = true;
            Queue(0).StallRestartEvent->Trigger(); Queue(1
            ).StallRestartEvent->Trigger();
        }
        else
        {
            checkThreadGraph(Queue(QueueIndex).StallRestartEvent);// make sure we are
started    up
            Queue(QueueIndex).QuitForReturn =true;
        }
    }

    virtualbool EnqueueFromOtherThread(int32 QueueIndex, FBaseGraphTask* Task) override {

        TestRandomizedThreads();
        checkThreadGraph(Task && Queue(QueueIndex).StallRestartEvent);// make sure we are started up


        uint32 PriIndex = ENamedThreads::GetTaskPriority(Task->ThreadToExecuteOn)?0:1; int32 ThreadToStart =
        Queue(QueueIndex).StallQueue.Push(Task, PriIndex);

        if(ThreadToStart >=0) {

            QUICK_SCOPE_CYCLE_COUNTER(STAT_TaskGraph_EnqueueFromOtherThread_Trigger);
            checkThreadGraph(ThreadToStart ==0);TASKGRAPH_SCOPE_CYCLE_COUNTER(1,

STAT_TaskGraph_EnqueueFromOtherThread_Trigger);
            Queue(QueueIndex).StallRestartEvent->Trigger(); returntrue;
```

```
            }
            returnfalse;
        }

        virtualbool IsProcessingTasks(int32 QueueIndex) override {

            return!!Queue(QueueIndex).RecursionGuard;
        }

    private:
        //Thread task queue.
        structFThreadTaskQueue {

            FStallingTaskQueue<FBaseGraphTask, PLATFORM_CACHE_LINE_SIZE,2> StallQueue;//The blocked task queue.


            uint32 RecursionGuard;//Prevents circular (recursive) calls. boolQuitForReturn;//
            Whether to request to exit. boolQuitForShutdown;//Whether to request
            shutdown. FEvent* StallRestartEvent;//Blocking event when the thread is fully
            loaded.
        };

        FORCEINLINE FThreadTaskQueue&Queue(int32 QueueIndex) {

            checkThreadGraph(QueueIndex >=0&& QueueIndex < ENamedThreads::NumQueues); return
            Queues[QueueIndex];
        }
        FORCEINLINEconstFThreadTaskQueue&Queue(int32 QueueIndex)const {

            checkThreadGraph(QueueIndex >=0&& QueueIndex < ENamedThreads::NumQueues); return
            Queues[QueueIndex];
        }

        FThreadTaskQueue Queues[ENamedThreads::NumQueues];//A named thread-specific task queue.
    };
```

FTaskThreadAnyThreadis used to process tasks of unnamed threads. Since there are many unnamed threads, the processing of tasks is different from that of FNamedTaskThread:

```
classFTaskThreadAnyThread: public FTaskThreadBase {

public:
    virtualvoid ProcessTasksUntilQuit(int32 QueueIndex) override {

        if(PriorityIndex != (ENamedThreads::BackgroundThreadPriority >>
        ENamedThreads::ThreadPriorityShift))
        {
            FMemory::SetupTLSCachesOnCurrentThread();
        }
        check(!QueueIndex);
        do
        {
            //Processing tasks
            ProcessTasks();
        }while(!Queue.QuitForShutdown && FPlatformProcess::SupportsMultithreading());// @Hack - quit now when
running with only one thread.
```

```
    }

    virtual uint64ProcessTasksUntilIdle(int32 QueueIndex) override {

        if(!FPlatformProcess::SupportsMultithreading()) {

            //Processing tasks
            returnProcessTasks();
        }
        else
        {
            check(0);
            return0;
        }
    }

    (......)

private:

#if  UE_EXTERNAL_PROFILING_ENABLED
    staticinlineconstTCHAR*ThreadPriorityToName(int32 PriorityIdx) {

        PriorityIdx <<= ENamedThreads::ThreadPriorityShift; if(PriorityIdx ==
        ENamedThreads::HighThreadPriority) {

            returnTEXT("Task Thread HP");//High priority worker thread
        }
        else if(PriorityIdx == ENamedThreads::NormalThreadPriority) {

            returnTEXT("Task Thread NP");//Normal priority worker thread
        }
        else if(PriorityIdx == ENamedThreads::BackgroundThreadPriority) {

            returnTEXT("Task Thread BP");//Background priority worker threads
        }
        else
        {
            returnTEXT("Task Thread Unknown Priority");
        }
    }
#endif


//The processing tasks here are similar toFNamedTaskThreadThere is a difference, in that the way to obtain tasks is different, fromTaskGraphThe unnamed task queue in the system gets the task.
    uint64ProcessTasks()
    {
        LLM_SCOPE(ELLMTag::TaskGraphTasksMisc);

        TStatId StallStatId;
        boolbCountAsStall =true; uint64
        ProcessedTasks =0;

        (......)

        verify(++Queue.RecursionGuard ==1); bool
        bDidStall =false; while(1)
```

```
        {
            //fromTaskGraphThe unnamed task queue in the system
            gets the task. FBaseGraphTask* Task = FindWork(); if(!
            Task)
            {
                (......)

                TestRandomizedThreads();
                if(FPlatformProcess::SupportsMultithreading()) {

                    FScopeCycleCounterScope(StallStatId); Queue.StallRestartEvent-
                    >Wait(MAX_uint32, bCountAsStall); bDidStall =true;

                }
                if(Queue.QuitForShutdown || !FPlatformProcess::SupportsMultithreading()) {

                    break;
                }
                TestRandomizedThreads();

                (......)

                continue;
            }
            TestRandomizedThreads();

            (......)

            bDidStall =false; Task-
            >Execute(NewTasks,                ENamedThreads::Type(ThreadId));
            ProcessedTasks++;
            TestRandomizedThreads(); if
            (Queue.bStallForTuning) {

                {
                    FScopeLockLock(&Queue.StallForTuning);
                }
            }
        }
        verify(!--Queue.RecursionGuard); return
        ProcessedTasks;
}

//Task queue data.
structFThreadTaskQueue {

        FEvent*    StallRestartEvent;
        uint32    RecursionGuard;
        bool QuitForShutdown;
        bool bStallForTuning;

        FCriticalSection StallForTuning;//Blocking critical section
};

//fromTaskGraphGet tasks from the
system. FBaseGraphTask*FindWork() {

        returnFTaskGraphImplementation::Get().FindWork(ThreadId);
```

```
    }

    FThreadTaskQueue Queue;//Task queue, only the first one is used for unnamed threads.

    int32 PriorityIndex;
};
```

- **ENamedThreads**

Before understanding the implementation and use of TaskGraph, it is necessary to understand the mechanism related to ENamedThreads. ENamedThreads is a namespace that provides operations for encoding and decoding threads and priorities. Its declaration and analysis are as follows:

```
namespace ENamedThreads {

    enumType: int32 {

        UnusedAnchor =  - 1,

        //----Dedicated (named) threads----
#if  STATS
        StatsThread,    //  Statistics thread
#endif
        RHIThread,      //  RHIThreads
        AudioThread,    //  Audio Thread
        GameThread,     //  Game Thread
        ActualRenderingThread = GameThread +1,//The actual rendering thread.GetRenderingThread()The rendering obtained may be the actual rendering thread or the game thread.

        AnyThread =0xff,//Any thread (unknown thread, unnamed thread)

        //----Queue index and priority----
        MainQueue =             0x000,  //Main Queue
        LocalQueue  =           0x100,//  Local Queue

        NumQueues =             2,
        ThreadIndexMask  =      0xff,
        QueueIndexMask  =       0x100,
        QueueIndexShift  =      8,

        //----Queue task index, priority----
        NormalTaskPriority  =   0x000,//  Normal task priority
        HighTaskPriority =              0x200,  //High task priority

        NumTaskPriorities  =    2,
        TaskPriorityMask  =     0x200,
        TaskPriorityShift  =    9,

        //----Thread Priority----
        NormalThreadPriority =0x000,//Normal thread priority
        HighThreadPriority =0x400,              //High thread priority
        BackgroundThreadPriority=0x800,//Background thread priority

        NumThreadPriorities  =  3,
        ThreadPriorityMask  =   0xC00,
        ThreadPriorityShift  =  10,
```

```
            //Combining Marks
#if    STATS
            StatsThread_Local = StatsThread | LocalQueue,
#endif
            GameThread_Local = GameThread | LocalQueue, ActualRenderingThread_Local =
            ActualRenderingThread | LocalQueue,

            AnyHiPriThreadNormalTask = AnyThread | HighThreadPriority | NormalTaskPriority,
            AnyHiPriThreadHiPriTask = AnyThread | HighThreadPriority | HighTaskPriority,

            AnyNormalThreadNormalTask = AnyThread | NormalThreadPriority | NormalTaskPriority,
            AnyNormalThreadHiPriTask = AnyThread | NormalThreadPriority | HighTaskPriority,

            AnyBackgroundThreadNormalTask = AnyThread | BackgroundThreadPriority |
            NormalTaskPriority,
            AnyBackgroundHiPriTask = AnyThread | BackgroundThreadPriority | HighTaskPriority,
      };

      structFRenderThreadStatics {

      private:
            //The rendering thread is stored. Note that it is an atomic operation type.
            staticCORE_API TAtomic<Type> RenderThread; staticCORE_API
            TAtomic<Type> RenderThread_Local;
      };

      //----Set and get the rendering thread interface----
      Type    GetRenderThread();
      Type    GetRenderThread_Local(); SetRenderThread
      void    (Type Thread); SetRenderThread_Local(Type
      void    Thread);

      externCORE_API int32 bHasBackgroundThreads;                    //Is there a background thread?
      externCORE_API int32 bHasHighPriorityThreads;//Is there a high priority thread?

      //----Set and get thread index, thread priority, task priority interface---- Type
            GetThreadIndex(Type       ThreadAndIndex);
      int32   GetQueueIndex(Type      ThreadAndIndex);
      int32   GetTaskPriority(Type          ThreadAndIndex);
      int32   GetThreadPriorityIndex(Type             ThreadAndIndex);

      TypeSetPriorities(Type ThreadAndIndex, Type ThreadPriority, Type TaskPriority); TypeSetPriorities(Type
      ThreadAndIndex, int32 PriorityIndex,boolbHiPri); TypeSetThreadPriority(Type ThreadAndIndex, Type
      ThreadPriority); TypeSetTaskPriority(Type ThreadAndIndex, Type TaskPriority);

}
```

- **FTaskGraphInterface**

Many task types are mentioned above, but this section really involves the manager and factory FTaskGraphInterface of these tasks. FTaskGraphInterface is the manager of the task graph and provides the operation interface of the task:

```
classFTaskGraphInterface {
```

```
    virtualvoid QueueTask(class FBaseGraphTask* Task, ENamedThreads::Type ThreadToExecuteOn,
ENamedThreads::Type CurrentThreadIfKnown = ENamedThreads::AnyThread) = 0;


public:
    // FTaskGraphInterfaceObject operation interface
    staticCORE_APIvoid Startup(int32 NumThreads); staticCORE_API
    void Shutdown(); staticCORE_APIbool IsRunning(); static
    CORE_API FTaskGraphInterface&Get();


    //Thread operation interface.
    virtual ENamedThreads::TypeGetCurrentThreadIfKnown(boolbLocalQueue =false) =0; virtual int32
    GetNumWorkerThreads() =0;
    virtualbool IsThreadProcessingTasks(ENamedThreads::Type ThreadToCheck) =0; virtualvoid
    AttachToThread(ENamedThreads::Type CurrentThread)=0; virtual uint64ProcessThreadUntilIdle
    (ENamedThreads::Type CurrentThread)=0;


    //Task operation interface.
    virtualvoid ProcessThreadUntilRequestReturn(ENamedThreads::Type CurrentThread)=0; virtualvoid
    RequestReturn(ENamedThreads::Type CurrentThread)=0;
    virtualvoid WaitUntilTasksComplete(constFGraphEventArray& Tasks, ENamedThreads::Type CurrentThreadIfKnown =
ENamedThreads::AnyThread)=0;
    virtualvoid TriggerEventWhenTasksComplete(FEvent* InEvent,constFGraphEventArray& Tasks,
ENamedThreads::Type CurrentThreadIfKnown = ENamedThreads::AnyThread, ENamedThreads::Type TriggerThread =
ENamedThreads::AnyHiPriThreadHiPriTask)=0;
    void WaitUntilTaskCompletes(constFGraphEventRef& Task, ENamedThreads::Type
CurrentThreadIfKnown = ENamedThreads::AnyThread);
    void TriggerEventWhenTaskCompletes(FEvent* InEvent,constFGraphEventRef& Task, ENamedThreads::Type
CurrentThreadIfKnown = ENamedThreads::AnyThread, ENamedThreads::Type TriggerThread =
ENamedThreads::AnyHiPriThreadHiPriTask);
    virtualvoid AddShutdownCallback(TFunction<void()>&Callback) =0; static void
    BroadcastSlow_OnlyUseForSpecialPurposes(boolbDoTaskThreads,bool bDoBackgroundThreads, TFunction<
void(ENamedThreads::Type CurrentThread)>& Callback); };
```

The implementation of FTaskGraphInterface is in the FTaskGraphImplementation class. FTaskGraphImplementation uses a special thread object WorkerThreads as the execution carrier. Of course, if it is a dedicated thread (named thread, such as GameThread, RHI, ActualRenderingThread), it will enter a dedicated task queue. Since its implementation details are many, we will discuss them later.

- **FTaskGraphImplementation**

FTaskGraphImplementation inherits and implements the interface of FTaskGraphInterface. Some interfaces and implementations are as follows:

```
// Engine\Source\Runtime\Core\Private\Async\TaskGraph.cpp

classFTaskGraphImplementation: public FTaskGraphInterface {

public:
    staticFTaskGraphImplementation&Get();


    //Constructor, calculate the number of task threads, create dedicated threads and unnamed threads, etc.
```

```
FTaskGraphImplementation(int32) {

        bCreatedHiPriorityThreads =             !!ENamedThreads::bHasHighPriorityThreads;
        bCreatedBackgroundPriorityThreads               = !!ENamedThreads::bHasBackgroundThreads;

        int32 MaxTaskThreads = MAX_THREADS;//The default maximum number of task threads is83.
        int32 NumTaskThreads = FPlatformMisc::NumberOfWorkerThreadsToSpawn();//Get the number of task threads based on the number
of hardware cores.

        // Handles platforms that cannot support multithreading.
        if (!FPlatformProcess::SupportsMultithreading())
        {
                MaxTaskThreads = 1; NumTaskThreads =
                1;
                LastExternalThread          = (ENamedThreads::Type)
(ENamedThreads::ActualRenderingThread           - 1);
bCreatedHiPriorityThreads                       =false;
bCreatedBackgroundPriorityThreads                               =  false;
ENamedThreads::bHasBackgroundThreads                                =  0;
ENamedThreads::bHasHighPriorityThreads                                 =0;
        }
        else
        {
                LastExternalThread = ENamedThreads::ActualRenderingThread;
        }

        //Number of dedicated threads
        NumNamedThreads = LastExternalThread +1;
        //Calculating the number of worker thread sets is related to whether high-priority threads are enabled and
        whether background-priority threads are created. NumTaskThreadSets =1+ bCreatedHiPriorityThreads +
        bCreatedBackgroundPriorityThreads;
        //Calculate the number of task threads actually needed, the maximum number is no more than83indivual.
        NumThreads = FMath::Max<int32>(FMath::Min<int32>(NumTaskThreads *

NumTaskThreadSets + NumNamedThreads, MAX_THREADS), NumNamedThreads +1);
        NumThreads = FMath::Min(NumThreads, NumNamedThreads + NumTaskThreads *
        NumTaskThreadSets);

        NumTaskThreadsPerSet = (NumThreads - NumNamedThreads) / NumTaskThreadSets;

        ReentrancyCheck.Increment();// just checking for reentrancy PerThreadIDTLSSlot
        = FPlatformTLS::AllocTlsSlot();

        //Create all task threads.
        for(int32 ThreadIndex =0; ThreadIndex < NumThreads; ThreadIndex++) {

                check(!WorkerThreads[ThreadIndex].bAttached);// reentrant? //Create threads separately depending on
                whether they are dedicated threads.
                boolbAnyTaskThread = ThreadIndex >= NumNamedThreads; if
                (bAnyTaskThread)
                {
                        WorkerThreads[ThreadIndex].TaskGraphWorker = new
FTaskThreadAnyThread(ThreadIndexToPriorityIndex(ThreadIndex));
                }
                else
                {
                        WorkerThreads[ThreadIndex].TaskGraphWorker = new FNamedTaskThread;
                }
                WorkerThreads[ThreadIndex].TaskGraphWorker-
```

```
> Setup(ENamedThreads::Type(ThreadIndex), PerThreadIDTLSSlot, &WorkerThreads[ThreadIndex]); }


        TaskGraphImplementationSingleton = this;//Assignmentthisarrive
TaskGraphImplementationSingleton,so that it can be accessed externally.

        //Sets the properties of an unnamed thread.
        for(int32 ThreadIndex = LastExternalThread +1; ThreadIndex < NumThreads; ThreadIndex++)

        {
            FString Name;
            constANSICHAR* GroupName ="TaskGraphNormal";
            int32 Priority = ThreadIndexToPriorityIndex(ThreadIndex); EThreadPriority
            ThreadPri;
            uint64 Affinity = FPlatformAffinity::GetTaskGraphThreadMask(); if(Priority ==1) {


                Name = FString::Printf(TEXT("TaskGraphThreadHP %d"), ThreadIndex -
(LastExternalThread +1));
                GroupName ="TaskGraphHigh";
                ThreadPri = TPri_SlightlyBelowNormal;// we want even hi priority tasks
below the normal threads

                // If the platform defines
FPlatformAffinity::GetTaskGraphHighPriorityTaskMask then use it
                if(FPlatformAffinity::GetTaskGraphHighPriorityTaskMask()                    !=
0xFFFFFFFFFFFFFFFF)
                {
                    Affinity = FPlatformAffinity::GetTaskGraphHighPriorityTaskMask();
                }
            }
            else if(Priority ==2) {

                Name = FString::Printf(TEXT("TaskGraphThreadBP %d"), ThreadIndex -
(LastExternalThread +1));
                GroupName = "TaskGraphLow";
                ThreadPri      = TPri_Lowest;
                // If the platform defines
FPlatformAffinity::GetTaskGraphBackgroundTaskMask then use it
                if(FPlatformAffinity::GetTaskGraphBackgroundTaskMask() !=
0xFFFFFFFFFFFFFFFF)
                {
                    Affinity = FPlatformAffinity::GetTaskGraphBackgroundTaskMask();
                }
            }
            else
            {
                Name = FString::Printf(TEXT("TaskGraphThreadNP %d"), ThreadIndex -
(LastExternalThread +1));
                ThreadPri = TPri_BelowNormal;// we want normal tasks below normal threads
like the game thread
            }

            //Calculate thread stack size.
#if    WITH_EDITOR
            uint32 StackSize =1024*1024;
#elif( UE_BUILD_SHIPPING || UE_BUILD_TEST )
            uint32 StackSize =384*1024;
```

```
#else
                uint32 StackSize =512*1024;
#endif
                //The execution thread that actually creates the worker thread.
                WorkerThreads[ThreadIndex].RunnableThread            =
FRunnableThread::Create(&Thread(ThreadIndex), *Name, StackSize, ThreadPri, Affinity);// these are below normal
threads so that they sleep when the named threads are active
                WorkerThreads[ThreadIndex].bAttached =true;

                if    (WorkerThreads[ThreadIndex].RunnableThread)
                {
                    TRACE_SET_THREAD_GROUP(WorkerThreads[ThreadIndex].RunnableThread-
> GetThreadID(),      GroupName);
                }
            }
        }

        //Team entry task.
        virtualvoid QueueTask(FBaseGraphTask* Task, ENamedThreads::Type ThreadToExecuteOn,
ENamedThreads::Type InCurrentThreadIfKnown = ENamedThreads::AnyThread) final override
        {
            TASKGRAPH_SCOPE_CYCLE_COUNTER(2, STAT_TaskGraph_QueueTask);

            if(ENamedThreads::GetThreadIndex(ThreadToExecuteOn) == ENamedThreads::AnyThread) {

                TASKGRAPH_SCOPE_CYCLE_COUNTER(3, //    STAT_TaskGraph_QueueTask_AnyThread);
                    Processing with multithreading support.
                if    (FPlatformProcess::SupportsMultithreading())
                {
                    //Processing priority.
                    uint32 TaskPriority = ENamedThreads::GetTaskPriority(Task-
>ThreadToExecuteOn);
                    int32 Priority = ENamedThreads::GetThreadPriorityIndex(Task-
>ThreadToExecuteOn);
                    if(Priority == (ENamedThreads::BackgroundThreadPriority >>
ENamedThreads::ThreadPriorityShift) && (!bCreatedBackgroundPriorityThreads || !
ENamedThreads::bHasBackgroundThreads))
                    {
                        Priority = ENamedThreads::NormalThreadPriority >>
ENamedThreads::ThreadPriorityShift;// we don't have background threads, promote to normal
                        TaskPriority = ENamedThreads::NormalTaskPriority >>
ENamedThreads::TaskPriorityShift;// demote to normal task pri
                    }
                    else if(Priority == (ENamedThreads::HighThreadPriority >>
ENamedThreads::ThreadPriorityShift) && (!bCreatedHiPriorityThreads || !
ENamedThreads::bHasHighPriorityThreads))
                    {
                        Priority = ENamedThreads::NormalThreadPriority >>
ENamedThreads::ThreadPriorityShift;// we don't have hi priority threads, demote to normal
                        TaskPriority = ENamedThreads::HighTaskPriority >>
ENamedThreads::TaskPriorityShift;// promote to hi task pri
                    }

                    uint32 PriIndex = TaskPriority?0:1; check(Priority >=0&& Priority <
                    MAX_THREAD_PRIORITIES); {

                        TASKGRAPH_SCOPE_CYCLE_COUNTER(4,
STAT_TaskGraph_QueueTask_IncomingAnyThreadTasks_Push);
```

```
                        //Push the task into the queue to be executed, and obtain and execute the executable task index (maybe none).
                        int32 IndexToStart = IncomingAnyThreadTasks[Priority].Push(Task,
PriIndex);
                        if(IndexToStart >=0) {

                            StartTaskThread(Priority, IndexToStart);
                        }
                    }
                    return;
                }
                else
                {
                    ThreadToExecuteOn      = ENamedThreads::GameThread;
                }
            }

            //The following are processes that do not support multithreading.
            ENamedThreads::Type CurrentThreadIfKnown;
            if(ENamedThreads::GetThreadIndex(InCurrentThreadIfKnown)                    ==
            ENamedThreads::AnyThread)
            {
                CurrentThreadIfKnown       =  GetCurrentThread();
            }
            else
            {
                CurrentThreadIfKnown       = ENamedThreads::GetThreadIndex(InCurrentThreadIfKnown);
                checkThreadGraph(CurrentThreadIfKnown            ==
ENamedThreads::GetThreadIndex(GetCurrentThread()));
            }
            {
                int32 QueueToExecuteOn = ENamedThreads::GetQueueIndex(ThreadToExecuteOn);
                ThreadToExecuteOn = ENamedThreads::GetThreadIndex(ThreadToExecuteOn); FTaskThreadBase*
                Target = &Thread(ThreadToExecuteOn);
                if(ThreadToExecuteOn == ENamedThreads::GetThreadIndex(CurrentThreadIfKnown)) {

                    Target->EnqueueFromThisThread(QueueToExecuteOn,              Task);
                }
                else
                {
                    Target->EnqueueFromOtherThread(QueueToExecuteOn,              Task);
                }
            }
        }

        virtual            int32GetNumWorkerThreads() final override;
        virtual ENamedThreads::TypeGetCurrentThreadIfKnown(boolbLocalQueue) final override; virtualbool
        IsThreadProcessingTasks(ENamedThreads::Type ThreadToCheck) final override;


        //Import the current thread into the specifiedIndex.
        virtualvoid AttachToThread(ENamedThreads::Type CurrentThread) final override;

        //----Processing task interface----

        virtual uint64ProcessThreadUntilIdle(ENamedThreads::Type CurrentThread) final override;

        virtualvoid ProcessThreadUntilRequestReturn(ENamedThreads::Type CurrentThread) final override;
```

```
        virtualvoid RequestReturn(ENamedThreads::Type CurrentThread) final override; virtualvoid WaitUntilTasksComplete(
        constFGraphEventArray& Tasks, ENamedThreads::Type CurrentThreadIfKnown = ENamedThreads::AnyThread) final
        override;
        virtualvoid TriggerEventWhenTasksComplete(FEvent* InEvent,constFGraphEventArray& Tasks,
ENamedThreads::Type CurrentThreadIfKnown = ENamedThreads::AnyThread, ENamedThreads::Type TriggerThread =
ENamedThreads::AnyHiPriThreadHiPriTask) final override;

        virtualvoid AddShutdownCallback(TFunction<void()>& Callback);

        //----Task scheduling interface----

        //Start a task thread with the specified priority and index.
        void StartTaskThread(int32 Priority, int32 IndexToStart); void
        StartAllTaskThreads(boolbDoBackgroundThreads); FBaseGraphTask*FindWork
        (ENamedThreads::Type ThreadInNeed); void StallForTuning(int32 Index,bool
        Stall); void SetTaskThreadPriorities(EThreadPriority Pri);


private:
        //Get the task thread reference of the specified index.
        FTaskThreadBase&Thread(int32 {          Index)

                checkThreadGraph(Index >=0&& Index < NumThreads);
                checkThreadGraph(WorkerThreads[Index].TaskGraphWorker->GetThreadId() return            ==  Index);
                *WorkerThreads[Index].TaskGraphWorker;
        }

        //Get the current thread index.
        ENamedThreads::TypeGetCurrentThread(); int32
        ThreadIndexToPriorityIndex(int32                    ThreadIndex);

        enum
        {
                MAX_THREADS =26* (CREATE_HIPRI_TASK_THREADS + CREATE_BACKGROUND_TASK_THREADS + 1) +
ENamedThreads::ActualRenderingThread +1, MAX_THREAD_PRIORITIES =3

        };

        FWorkerThread               WorkerThreads[MAX_THREADS];       //An array of all worker thread (task thread) objects.
        int32       int32           NumThreads;             //  The number of threads actually used.
        int32 int32                 NumNamedThreads;    // The number of dedicated threads.
                                    NumTaskThreadSets;//    The number of task threads to collect.
                                    NumTaskThreadsPerSet;       //The number of task threads each collection has.

        bool                    bCreatedHiPriorityThreads;
        bool                    bCreatedBackgroundPriorityThreads;

        ENamedThreads::Type     LastExternalThread;
        FThreadSafeCounter          ReentrancyCheck;
        uint32                      PerThreadIDTLSSlot;

        TArray<TFunction<void()> > ShutdownCallbacks;//Callback before destruction.

        FStallingTaskQueue<FBaseGraphTask, PLATFORM_CACHE_LINE_SIZE,2>
        IncomingAnyThreadTasks[MAX_THREAD_PRIORITIES];
};
```

In summary, TaskGraph creates different sets of worker threads based on thread priority and whether background threads are enabled, and then creates their FWorkerThread objects. When a task is queued, it is pushed to the task list IncomingAnyThreadTasks (type is FStallingTaskQueue, a thread-safe, lock-free linked list), and the executable task index is taken out. The corresponding worker thread is enabled to execute according to the task's attributes (which thread you want to execute, priority, task index).

The working thread FWorkerThread involved in TaskGraph is declared as follows:

```
structFWorkerThread
{
	FTaskThreadBase*	TaskGraphWorker;	//	LocatedFTaskThreadObject (beingFTaskThreadThe object owns the runnable
	FRunnableThread*	RunnableThread;		//	thread that actually executes the task.
	bool
	bAttached;//Whether to attach the thread. (Generally used for dedicated threads)
};
```

It can be seen that TaskGraph also uses FRunnableThread to execute tasks. The TaskGraph system is finally connected with FRunnableThread to form a closed loop.

At this point, the main outline of the TaskGraph system has finally been explained. Of course, there are still many technical details (such as synchronization events, triggering details, scheduling algorithms, lock-free linked lists, and some concepts) that have not been involved. These are left for readers to study the UE source code and explore.

2.5 UE's multi-threaded rendering

After a lot of basic preparation, we finally get back to the topic and talk about the knowledge related to UE's multi-threaded rendering.

2.5.1 UE's multi-threaded rendering basics

2.5.1.1 Main types of scene and rendering modules

UE's scene and rendering modules involve many concepts, the main types and analysis are as follows:

type	Analysis
UWorld	Contains a set of Actors and components that can interact with each other. Multiple levels can be loaded into or unloaded from UWorld. Multiple levels instances can exist at the same time.
ULevel	A level stores a set of Actors and components in the same file.

type	Analysis
USceneComponent	Scene components are the parent class of all objects that can be added to the scene, such as lights, models, fog, etc.
UPrimitiveComponent	The primitive component is the parent class of all objects that can be rendered or have physical simulation. It is the smallest granularity unit for CPU layer clipping.
ULightComponent	Light source component is the parent class of all light source types.
FScene	It is the representative of UWorld in the rendering module. Only objects added to FScene will be perceived by the renderer. The rendering thread has all the states of FScene (the game thread cannot modify it directly).
FPrimitiveSceneProxy	The primitive scene proxy is the representative of UPrimitiveComponent in the renderer, mirroring the state of UPrimitiveComponent in the rendering thread.
FPrimitiveSceneInfo	The internal state of the renderer (describing the implementation of FRendererModule), is equivalent to the fusion of UPrimitiveComponent and FPrimitiveSceneProxy. Only the renderer module exists, so the engine module cannot perceive its existence.
FSceneView	Describes a single view in FScene. The same FScene allows multiple views. In other words, a scene can be drawn by multiple views, or multiple views can be drawn at the same time. A new view instance is created for each frame.
FViewInfo	The view is represented inside the renderer. Only the renderer module exists, and the engine module is invisible.
FSceneViewState	Stores private information about the view's renderer that needs to be accessed across frames. In the Game instance, each ULocalPlayer has an FSceneViewState instance.
FSceneRenderer	It is created for each frame, encapsulating temporary data between frames. derives FDeferredShadingSceneRenderer (deferred shading scene renderer) and FMobileSceneRenderer (mobile scene renderer), representing the default renderers for PC and mobile respectively.

2.5.1.2 Engine module and rendering module representatives

In order to have a clear structure, reduce dependencies between modules, and speed up iteration, UE is divided into many modules, the most important of which are engine module, renderer module, core, RHI, plug-in, etc. The previous section mentioned many concepts and types, some of which exist in the engine module (Engine Module) and some exist in the renderer module (Renderer Module), as shown in the following table:

Engine Module	Renderer Module
UWorld	FScene
UPrimitiveComponent/FPrimitiveSceneProxy	FPrimitiveSceneInfo
FSceneView	FViewInfo
ULocalPlayer	FSceneViewState
ULightComponent/FLightSceneProxy	FLightSceneInfo

2.5.1.3 Game Thread and Rendering Thread Representatives

Objects in the game thread usually perform logical updates and have a persistent data in memory. In order to avoid competition between the game thread and the rendering thread, an additional memory copy is stored in the rendering thread, and a different type is used. The following is a common type mapping relationship in UE (game thread objects start with U, and rendering threads start with F):

Game Thread	Rendering Thread
UWorld	FScene
UPrimitiveComponent	FPrimitiveSceneProxy/FPrimitiveSceneInfo
-	FSceneView / FViewInfo
ULocalPlayer	FSceneViewState
ULightComponent	FLightSceneProxy/FLightSceneInfo

Game threads represent operations that are generally performed by the game thread, and rendering threads represent operations that are mainly performed by the rendering thread. If you attempt to operate data across threads, unpredictable results will occur and race conditions will occur.

```
/** SceneProxyWhen registered into the scene, it will be constructed and data will be passed in the game
thread. */ FStaticMeshSceneProxy::FStaticMeshSceneProxy(UStaticMeshComponent*                    InComponent):
    FPrimitiveSceneProxy(...), Owner(InComponent->GetOwner()) <========Here will
    AActorPointers are cached
    . . .
```

```
/** SceneProxyofDrawDynamicElementsWill be called by the renderer in the * /
void rendering thread FStaticMeshSceneProxy::DrawDynamicElements(...)
{
    if(Owner->AnyProperty) <==========will cause a race condition! state!! andUObjectThe The game thread hasAActor,UObjectAll
object may beGCIf it is dropped, accessing it again will cause the program to crash! !
}
```

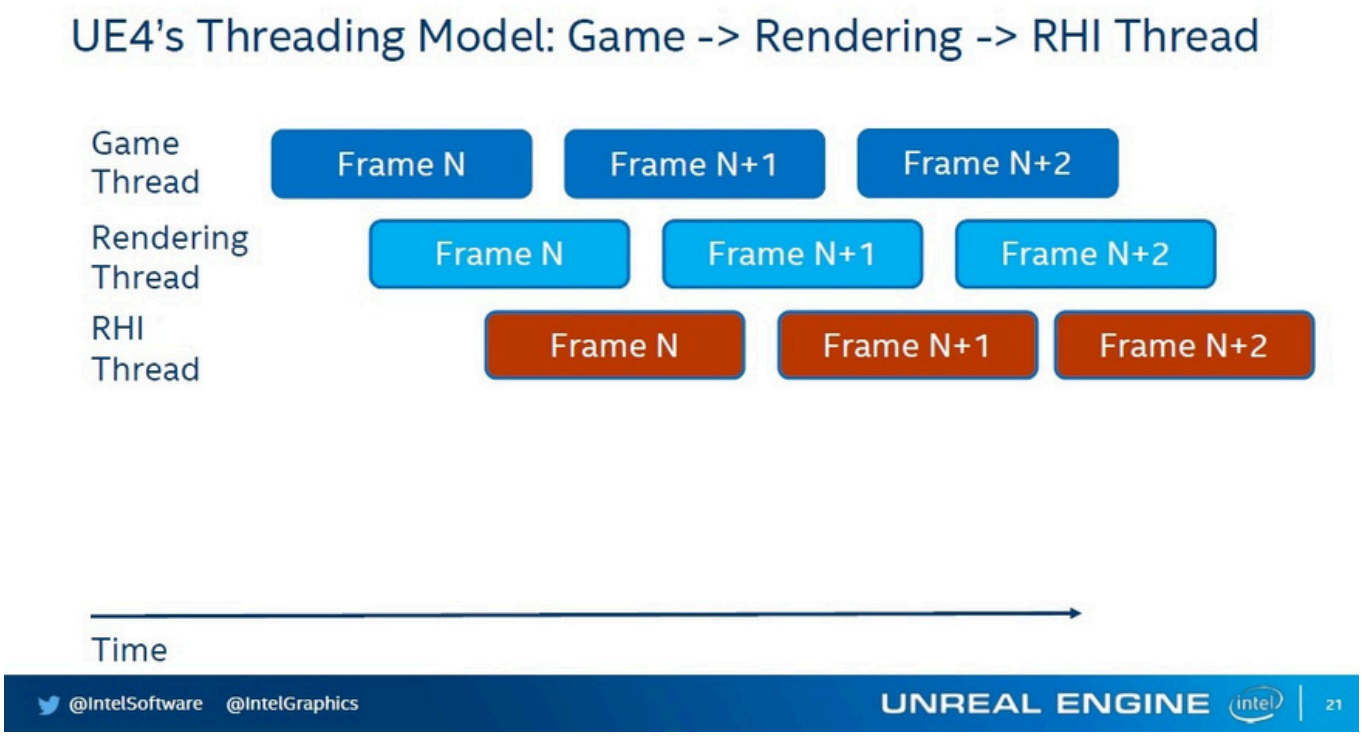
Some representatives are special, such as FPrimitiveSceneProxy and FLightSceneProxy. These scene proxies belong to the engine module, but they are also exclusive objects of the rendering thread, which means that they are the bridge connecting the game thread and the rendering thread, and are tools for transferring data between threads.

2.5.2 Overview of UE's multi-threaded rendering

By default, UE has a game thread (Game Thread), a rendering thread (Render Thread), and an RHI thread (RHI Thread), all of which run independently on a dedicated thread (FRunnableThread).

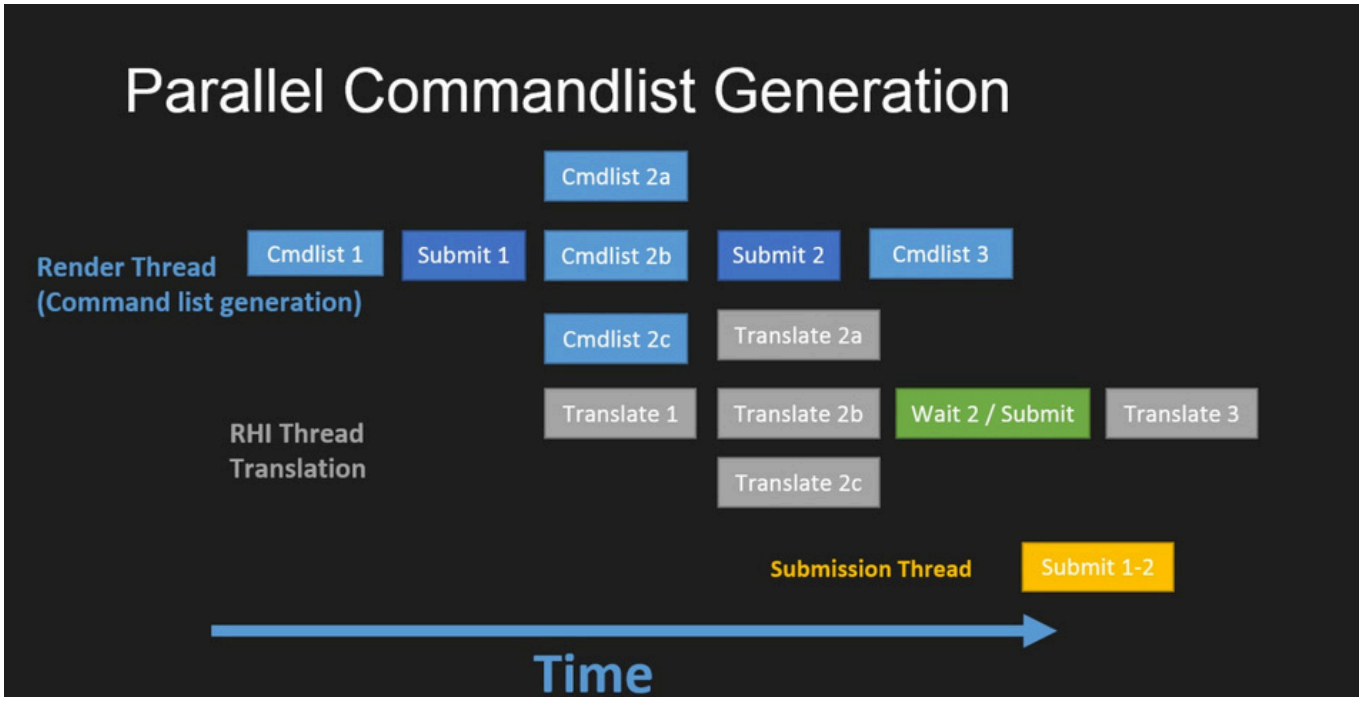
The game thread queues callback interfaces to the rendering thread's Queue through certain interfaces, so that when the rendering thread runs later, it gets callbacks from the rendering thread's Queue and executes them one by one, thus generating a Command List.

The Command List generated by the rendering thread as the frontend is platform-independent and is an abstract graphics API call; while the RHI thread as the backend will execute and convert the Command List of the rendering thread into a call to the specified graphics API (called Graphical Command) and submit it to the GPU for execution. The data processed by these threads are usually different frames, for example, the game thread processes N frame data, and the rendering thread and RHI thread process N-1 frame data.



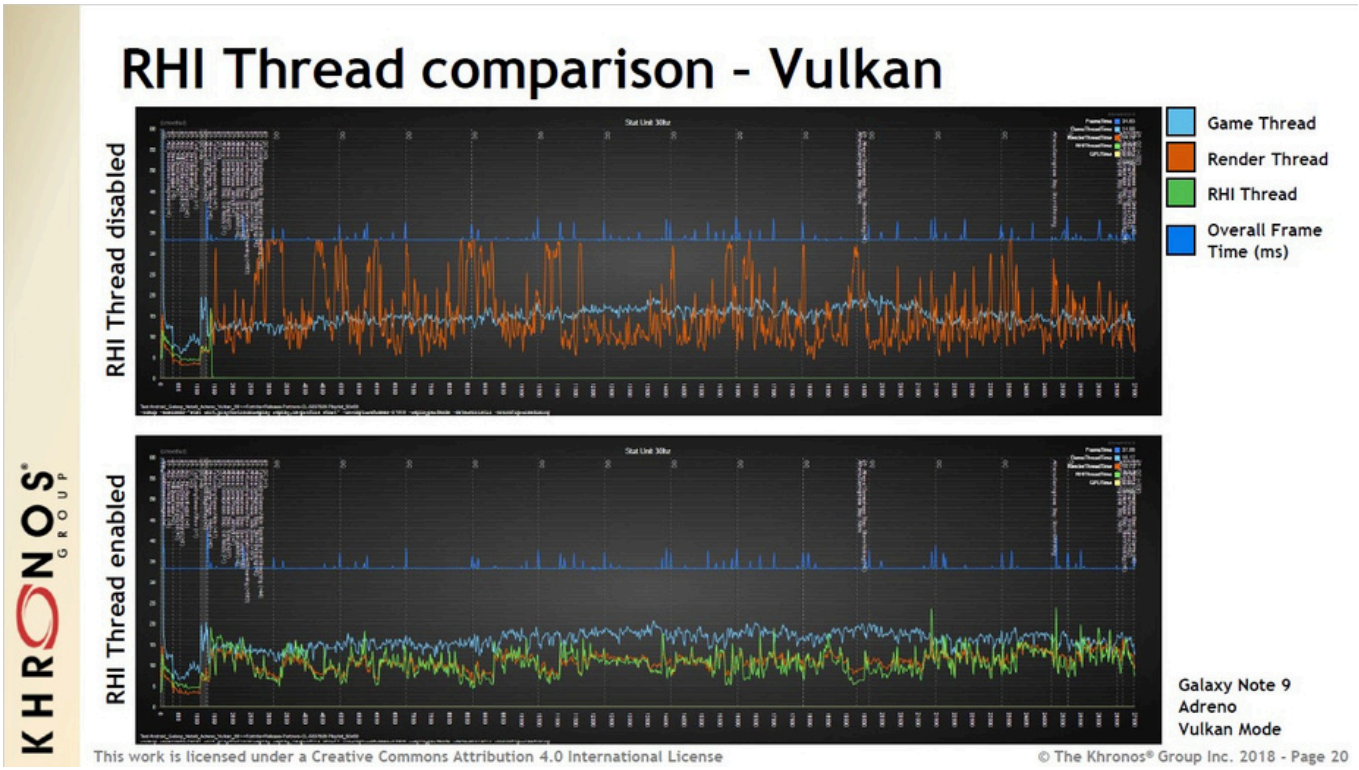
There are exceptions, such as the rendering thread and RHI thread running very fast, almost no delay, in this case, the game thread processes N frames, while the rendering thread may process N or N-1 frames, and the RHI thread may also convert N or N-1 frames. However, the rendering thread cannot lag behind the game thread by one frame, otherwise the game thread will be stuck until the rendering thread processes all instructions.

In addition, rendering instructions can be generated in parallel, and the RHI threads can also convert these instructions in parallel, as shown below:



UE4 generates command lists in parallel.

The benefits of enabling multi-threaded rendering are higher frame rates and lower frame rate changes (more stable frame rates). Take Fortnite mobile as an example. Before enabling the RHI thread, the rendering thread fluctuated sharply, but after adding the RHI thread, the fluctuations were much smoother, basically consistent with the game thread, and the frame rate was also greatly improved:



2.5.3 Implementation of Game Thread and Rendering Thread

2.5.3.1 Game Thread Implementation

The game thread is called the main thread. It is the heart of the engine, carrying the main game logic and running process, and is also the data initiator of other threads.

The game thread is created by running the program entry thread. It is created at the same time when the system starts the process (because the process needs at least one thread to work). It is directly stored in the global variable when the engine starts, and will be set to the TaskGraph system later:

```
// Engine\Source\Runtime\Launch\Private\LaunchEngineLoop.cpp

int32FEngineLoop::PreInitPreStartupScreen(constTCHAR* CmdLine) {

    (......)

    //Get the current threadid,     Stored in a global variable.
    GGameThreadId =     FPlatformTLS::GetCurrentThreadId();
    GIsGameThreadIdInitialized              =true;

    FPlatformProcess::SetThreadAffinityMask(FPlatformAffinity::GetMainGameMask()); //Set game thread data
    (but many platforms have empty implementations) FPlatformProcess::SetupGameThread();


    (......)

    if(bCreateTaskGraphAndThreadPools) {

        SCOPED_BOOT_TIMING("FTaskGraphInterface::Startup");
        FTaskGraphInterface::Startup(FPlatformMisc::NumberOfCores());
```

```
            //Attach the current thread (main thread) toTaskGraphofGameThreadNamed slot. This way the main thread andTaskGraphLinked up
Come.
            FTaskGraphInterface::Get().AttachToThread(ENamedThreads::GameThread);
        }
}
```

The above code also shows that**the main thread, game thread and TaskGraph system's ENamedThreads::GameThread are actually the same thing, they are all the same thread!**

After the above initialization and settings, other places can process tasks in parallel through the TaskGraph system, and can also access global variables to determine whether the game thread has been initialized and whether the current thread is a game thread:

```
bool IsInGameThread()
{
        returnGIsGameThreadIdInitialized && FPlatformTLS::GetCurrentThreadId() == GGameThreadId;

}
```

2.5.3.2 Implementation of rendering thread

The rendering thread is different from the game. It is an independent thread dedicated to generating rendering instructions and rendering logic. RenderingThread.h declares all external interfaces, some of which are as follows:

```
// Engine\Source\Runtime\RenderCore\Public\RenderingThread.h

//Whether to enable independent rendering thread.false,All rendering commands will be executed immediately instead of being put into the
rendering command queue. externRENDERCORE_APIboolGIsThreadedRendering;

//Whether a rendering thread should be created. Usually set by command line argument orToggleRenderingThreadConsole
parameter settings. externRENDERCORE_APIboolGUseThreadedRendering;

//Is it enabled?RHIThreads
externRENDERCORE_APIvoid SetRHIThreadEnabled(boolbEnableDedicatedThread,bool
bEnableRHIOnTaskThreads);

(.....)

//Start the rendering thread.
externRENDERCORE_APIvoid StartRenderingThread();

//Stop the rendering thread.
externRENDERCORE_APIvoid StopRenderingThread();

//Check if the rendering thread is healthy (whetherCrash),ifcrash,Then useUE_Log
Output log. externRENDERCORE_APIvoid CheckRenderingThreadHealth();

//Check if the rendering thread is healthy (whetherCrash
externRENDERCORE_APIbool IsRenderingThreadHealthy();

//Add a must before the next scene is drawn orflushTasks that are completed before rendering commands.
externRENDERCORE_APIvoid AddFrameRenderPrerequisite(constFGraphEventRef& TaskToAdd);
```

```
//Mobile phone frame rendering pre-task, ensuring that all rendering commands are queued.
externRENDERCORE_APIvoid AdvanceFrameRenderPrerequisite();


//Wait for all rendering threads' rendering commands to be executed. Will block the game thread and can only be called by the game thread.
externRENDERCORE_APIvoid FlushRenderingCommands(boolbFlushDeferredDeletes =false);

externRENDERCORE_APIvoid FlushPendingDeleteRHIResources_GameThread(); extern
RENDERCORE_APIvoid FlushPendingDeleteRHIResources_RenderThread();

externRENDERCORE_APIvoid TickRenderingTickables();

externRENDERCORE_APIvoid StartRenderCommandFenceBundler(); extern
RENDERCORE_APIvoid StopRenderCommandFenceBundler();

(.....)
```

RenderingThread.h also has a very important macroENQUEUE_RENDER_COMMAND,which is used to queue rendering instructions to the rendering thread. Here is its declaration and implementation:

```
//Enqueue rendering instructions to the rendering thread,TypeSpecifies the name of the rendering operation.
#defineENQUEUE_RENDER_COMMAND(Type)   \
 struct          Type##Name \
    {    \
        static const char* CStr() { return #Type; } \ static const TCHAR* TStr()
        { return TEXT(#Type); } \
    };   \
    EnqueueUniqueRenderCommand<Type##Name>
```

The last sentence above usesEnqueueUniqueRenderCommandthe command, continue tracking it:

```
//TSTRis the rendering command name,LAMBDAIs a callback
function. template<typename TSTR, typename LAMBDA>
FORCEINLINE_DEBUGGABLEvoid EnqueueUniqueRenderCommand(LAMBDA&& Lambda) {

    typedefTEnqueueUniqueRenderCommandType<TSTR, LAMBDA> EURCType;

    //   If you execute the callback directly in the rendering thread without enqueuing the
    if   rendering command. (IsInRenderingThread())
    {
        FRHICommandListImmediate& RHICmdList = GetImmediateCommandList_ForRenderCommand();
        Lambda(RHICmdList);
    }
    else
    {
        //   Need to be executed in a separate rendering thread
        if   (ShouldExecuteOnRenderThread())
        {
            CheckNotBlockedOnRenderThread();
            //fromGraphTaskCreates tasks and enqueues rendering commands when appropriate.


TGraphTask<EURCType>::CreateTask().ConstructAndDispatchWhenReady(Forward<LAMBDA>(Lambda));
        }
        else//If not executed in a separate rendering thread, it is executed directly. {

            EURCType TempCommand(Forward<LAMBDA>(Lambda));
```

```
                FScopeCycleCounterEURCMacro_Scope(TempCommand.GetStatId());
                TempCommand.DoTask(ENamedThreads::GameThread, FGraphEventRef());
            }
        }
}
```

The above shows that if there is an independent rendering thread, the rendering command will eventually be queued into the TaskGraph task queue, waiting for the right time to be executed in the rendering thread. Among themTEnqueueUniqueRenderCommandTypeis the special TaskGraph task type dedicated to rendering commands, which is declared as follows:

```
classRENDERCORE_API FRenderCommand {

public:
    //All rendering instructions must be executed in the rendering thread.
    staticENamedThreads::TypeGetDesiredThread() {

        check(!GIsThreadedRendering || ENamedThreads::GetRenderThread() !=
        ENamedThreads::GameThread);
        returnENamedThreads::GetRenderThread();
    }

    staticESubsequentsMode::TypeGetSubsequentsMode() {

        returnESubsequentsMode::FireAndForget;
    }
};

template<typename TSTR, typename LAMBDA>
classTEnqueueUniqueRenderCommandType: public FRenderCommand {

public:
    TEnqueueUniqueRenderCommandType(LAMBDA&& InLambda) : Lambda(Forward<LAMBDA>(InLambda))
{}

    //Executing task.
    void DoTask(ENamedThreads::Type CurrentThread,constFGraphEventRef&
    MyCompletionGraphEvent)
    {
        TRACE_CPUPROFILER_EVENT_SCOPE_ON_CHANNEL_STR(TSTR::TStr(), RenderCommandsChannel);
        FRHICommandListImmediate& RHICmdList = GetImmediateCommandList_ForRenderCommand();
        Lambda(RHICmdList);
    }

    (.....)

private:
    LAMBDA Lambda;//Cache rendering callback function.
};
```

To better understand the operation of enqueuing rendering commands, let's take a specific example, adding lights to the scene:

```
void FScene::AddLight(ULightComponent* Light) {

    (.....)

    // Send a command to the rendering thread to add the light to the scene. FScene* Scene = this;

    FLightSceneInfo* LightSceneInfo = Proxy->LightSceneInfo;

    //Rendering instructions are enqueued here so that the light data can be passed to the renderer on the
    rendering thread. ENQUEUE_RENDER_COMMAND(FAddLightCommand)(
        [Scene, LightSceneInfo](FRHICommandListImmediate& {        RHICmdList)

            CSV_SCOPED_TIMING_STAT_EXCLUSIVE(Scene_AddLight); FScopeCycleCounter
            Context(LightSceneInfo->Proxy->GetStatId()); Scene-
            >AddLightSceneInfo_RenderThread(LightSceneInfo);
        });
}
```

Substitute`ENQUEUE_RENDER_COMMAND(FAddLightCommand)`the previously parsed macros and templates and expand them. The complete code is as follows:

```
structFAddLightCommandName {

    static const char*CStr() {return"FAddLightCommand"; } static constTCHAR*TStr() {
    returnTEXT("FAddLightCommand"); }
};

EnqueueUniqueRenderCommand<FAddLightCommandName>(
    [Scene, LightSceneInfo](FRHICommandListImmediate& {        RHICmdList)

        CSV_SCOPED_TIMING_STAT_EXCLUSIVE(Scene_AddLight); FScopeCycleCounter
        Context(LightSceneInfo->Proxy->GetStatId()); Scene-
        >AddLightSceneInfo_RenderThread(LightSceneInfo);
    })
{
    typedefTEnqueueUniqueRenderCommandType<FAddLightCommandName, LAMBDA> EURCType;

    //  If you execute the callback directly in the rendering thread without enqueuing the
    if  rendering command. (IsInRenderingThread())
    {
        FRHICommandListImmediate& RHICmdList = GetImmediateCommandList_ForRenderCommand();
        Lambda(RHICmdList);
    }
    else
    {
        //  Need to be executed in a separate rendering thread
        if  (ShouldExecuteOnRenderThread())
        {
            CheckNotBlockedOnRenderThread();
            //fromGraphTaskCreates tasks and enqueues rendering commands when appropriate.

TGraphTask<EURCType>::CreateTask().ConstructAndDispatchWhenReady(Forward<LAMBDA>(Lambda));
        }
        elseIf not executed in a separate rendering thread, it is executed directly. {

            EURCType TempCommand(Forward<LAMBDA>(Lambda));
```

```
                FScopeCycleCounterEURCMacro_Scope(TempCommand.GetStatId());
                TempCommand.DoTask(ENamedThreads::GameThread, FGraphEventRef());
            }
        }
    }
```

FRenderingThread carries the main work of the rendering thread. Some of its interfaces and implementation codes are as follows:

```
// Engine\Source\Runtime\RenderCore\Private\RenderingThread.cpp

classFRenderingThread: public FRunnable {

private:
    boolbAcquiredThreadOwnership;                    //  When there is no independentRHIWhen the thread
                                                      The rendering thread will be captured by other threads.
public:
    FEvent* TaskGraphBoundSyncEvent;//TaskGraphSynchronize events so that the rendering thread can be bound to them before the main thread uses
themTaskGraphIn the system.

    FRenderingThread()
    {
        bAcquiredThreadOwnership //Get        =  false;
        synchronization events.
        TaskGraphBoundSyncEvent            = FPlatformProcess::GetSynchEventFromPool(true);
        RHIFlushResources();
    }

    // FRunnable interface. virtualbool Init(void)
    override {

        //Get the current threadIDTo global variablesGRenderThreadId,For reference elsewhere.
        GRenderThreadId = FPlatformTLS::GetCurrentThreadId();

        //    Handles thread capture relations.
        if   (!IsRunningRHIInSeparateThread())
        {
            bAcquiredThreadOwnership =true;
            RHIAcquireThreadOwnership();
        }

        returntrue;
    }

    (......)

    virtual uint32Run(void) override {

        //set upTLS.
        FMemory::SetupTLSCachesOnCurrentThread(); //Sets rendering
        thread platform-dependent data.
        FPlatformProcess::SetupRenderThread();

        (......)

        {
```

```
                //Enter the rendering thread main loop.
                RenderingThreadMain( TaskGraphBoundSyncEvent );
            }

            FMemory::ClearAndDisableTLSCachesOnCurrentThread(); return0;

        }
};
```

It can be seen that it will enter the rendering thread logic after running. Here we enter the RenderingThreadMain code to find out:

```
void RenderingThreadMain(FEvent* TaskGraphBoundSyncEvent) {

    LLM_SCOPE(ELLMTag::RenderingThreadMemory);

    //Set the render thread and local thread slots toActualRenderingThreadandActualRenderingThread_Local.
    ENamedThreads::Type RenderThread = ENamedThreads::Type(ENamedThreads::ActualRenderingThread);


    ENamedThreads::SetRenderThread(RenderThread);

ENamedThreads::SetRenderThread_Local(ENamedThreads::Type(ENamedThreads::ActualRenderingThr ead_Local));


    //Attach the current thread toTaskGraphofRenderThreadin the slot.
    FTaskGraphInterface::Get().AttachToThread(RenderThread);
    FPlatformMisc::MemoryBarrier();

    //Trigger a synchronization event, notifying the main thread that the rendering thread has been attached toTaskGraph,Ready
    to receive tasks. if( TaskGraphBoundSyncEvent !=NULL) {

        TaskGraphBoundSyncEvent->Trigger();
    }

    (......)

    //The different stages of processing in the rendering thread.
    FCoreDelegates::PostRenderingThreadCreated.Broadcast();
    check(GIsThreadedRendering);
    FTaskGraphInterface::Get().ProcessThreadUntilRequestReturn(RenderThread);
    FPlatformMisc::MemoryBarrier();
    check(!GIsThreadedRendering);
    FCoreDelegates::PreRenderingThreadDestroyed.Broadcast();

    (......)

    //Restore the thread to the game thread.
    ENamedThreads::SetRenderThread(ENamedThreads::GameThread);
    ENamedThreads::SetRenderThread_Local(ENamedThreads::GameThread_Local);
    FPlatformMisc::MemoryBarrier();
}
```

However, there is still a big question here, that is, FRenderingThread only gets the current thread as the rendering thread and attaches it to TaskGraph, but does not create a thread. So where is the

rendering thread created? Continuing to track, it turns out thatStartRenderingThread()the FRenderingThread instance is created in the interface, and its implementation code is as follows (excerpt):

```
// Engine\Source\Runtime\RenderCore\Private\RenderingThread.cpp

void StartRenderingThread() {

	(……)

	// Turn on the threaded rendering flag.
	GIsThreadedRendering =true;

	//createFRenderingThreadExamples. GRenderingThreadRunnable =
	new FRenderingThread();

	//Create rendering thread!!
	GRenderingThread = FRunnableThread::Create(GRenderingThreadRunnable,
*	BuildRenderingThreadName(ThreadCount),0,
FPlatformAffinity::GetRenderingThreadPriority(),
FPlatformAffinity::GetRenderingThreadMask(),
FPlatformAffinity::GetRenderingThreadFlags());

	(……)

	//Enables fences for rendering commands.
	FRenderCommandFence	Fence;
	Fence.BeginFence();
	Fence.Wait();

	(……)
}
```

If you continue tracing, you will find FEngineLoop::PreInitPostStartupScreen StartRenderingThread()that it is called in.

At this point, the creation, initialization and implementation of the main interfaces of the rendering thread have been analyzed.

2.5.3.3 Implementation of RHI Thread

The job of the RHI thread is to convert rendering instructions to the specified graphics API, create and upload rendering resources to the GPU. Its main logic is in FRHIThread, and the implementation code is as follows:

```
// Engine\Source\Runtime\RenderCore\Private\RenderingThread.cpp

classFRHIThread: public FRunnable {

public:
	FRunnableThread* Thread;		//LocatedRHIThreads.

	FRHIThread()
		: Thread(nullptr)
```

```
    {
        check(IsInGameThread());
    }

    void Start()
    {
        //Create at the beginningRHIThreads.
        Thread = FRunnableThread::Create(this, TEXT("RHIThread"),512*1024,
        FPlatformAffinity::GetRHIThreadPriority(),
            FPlatformAffinity::GetRHIThreadMask(), FPlatformAffinity::GetRHIThreadFlags() );

        check(Thread);
    }

    virtual uint32Run() override {

        LLM_SCOPE(ELLMTag::RHIMisc);

        //initializationTLS
        FMemory::SetupTLSCachesOnCurrentThread();
        //WillFRHIThreadLocatedRHIThread attached toaskGraphsystem and assigned toENamedThreads::RHIThread.
        FTaskGraphInterface::Get().AttachToThread(ENamedThreads::RHIThread); //start upRHIthread until the thread
        returns.

FTaskGraphInterface::Get().ProcessThreadUntilRequestReturn(ENamedThreads::RHIThread);
        //CleaningTLS.
        FMemory::ClearAndDisableTLSCachesOnCurrentThread(); return0;

    }

    //Singleton interface.
    staticFRHIThread&Get() {

        staticFRHIThread Singleton;//The use of local static variables can ensure thread safety.
        returnSingleton;
    }
};
```

It can be seen that the RHI thread is different from the rendering thread. It creates the actual thread directly in the FRHIThread object. The creation of FRHIThread is also inStartRenderingThread():

```
void StartRenderingThread() {

    (.....)

    if(GUseRHIThread_InternalUseOnly) {


FRHICommandListExecutor::GetImmediateCommandList().ImmediateFlush(EImmediateFlushType::Dis
patchToRHIThread);
        if(!FTaskGraphInterface::Get().IsThreadProcessingTasks(ENamedThreads::RHIThread)) {

            //createFRHIThreadinstance and
            start it. FRHIThread::Get().Start();
        }
        DECLARE_CYCLE_STAT(TEXT("Wait For RHIThread"), STAT_WaitForRHIThread,
```

```
STATGROUP_TaskGraphTasks);

//createRHIThe thread owner grabs the task, and lets the game thread
wait. FGraphEventRef CompletionEvent =
TGraphTask<FOwnershipOfRHIThreadTask>::CreateTask(NULL,
ENamedThreads::GameThread).ConstructAndDispatchWhenReady(true,
GET_STATID(STAT_WaitForRHIThread));
        QUICK_SCOPE_CYCLE_COUNTER(STAT_StartRenderingThread);
        //Let the game thread or local thread waitRHIThread handling (captures thread owners, most
        graphicsAPIEmpty) Completed.
FTaskGraphInterface::Get().WaitUntilTaskCompletes(CompletionEvent, ENamedThreads::GameThread_Local);
        //storageRHIThreadsid.
        GRHIThread_InternalUseOnly = FRHIThread::Get().Thread;
        check(GRHIThread_InternalUseOnly);
        GIsRunningRHIInDedicatedThread_InternalUseOnly                =true;
        GIsRunningRHIInSeparateThread_InternalUseOnly GRHIThread=Idt r=u e;
        GRHIThread_InternalUseOnly->GetThreadID();

        GRHICommandList.LatchBypass();
    }

    (......)
}
```

So how does the rendering thread enqueue tasks to the RHI thread? The answer is in RHICommandList.h:

```
// Engine\Source\Runtime\RHI\Public\RHICommandList.h

//RHICommand parent class
structFRHICommandBase
{
    FRHICommandBase* Next = nullptr;//Point to nextRHIOrder. //implementRHI
    Command and destroy.
    virtualvoid ExecuteAndDestruct(FRHICommandListBase& CmdList,
FRHICommandListDebugContext& DebugContext) =0; };


//RHICommand structure
template<typename TCmd, typename NameType = FUnnamedRhiCommand> struct
FRHICommand : public FRHICommandBase {

    (......)

    voidExecuteAndDestruct(FRHICommandListBase& CmdList, FRHICommandListDebugContext& Context)
override final
    {
        (......)

        TCmd *ThisCmd = static_cast<TCmd*>(this);

        ThisCmd->Execute(CmdList);
        ThisCmd->~TCmd();
    }
};
```

```
//TowardsRHIThread SendRHICommand macros.
#defineFRHICOMMAND_MACRO(CommandName) struct                                    \
PREPROCESSOR_JOIN(CommandName##String, {                __LINE__)                \
                                                                                \
        static const TCHAR* TStr() { return TEXT(#CommandName); }               \
};                                                                              \
struct CommandName final : public FRHICommand<CommandName,
PREPROCESSOR_JOIN(CommandName##String, __LINE__)>
```

The implementation mechanism of the RHI thread is similar to the rendering thread type, and is more concise. The following is a demonstration of its use:

```
// Engine\Source\Runtime\RHI\Public\RHICommandList.h
FRHICOMMAND_MACRO(FRHICommandDrawPrimitive) {

        uint32     BaseVertexIndex;
        uint32     NumPrimitives;
        uint32     NumInstances;

        FORCEINLINE_DEBUGGABLEFRHICommandDrawPrimitive(uint32 InBaseVertexIndex, uint32
InNumPrimitives, uint32 InNumInstances)
                :BaseVertexIndex(InBaseVertexIndex) ,
                   NumPrimitives(InNumPrimitives)
                , NumInstances(InNumInstances)
        { }
        RHI_APIvoid Execute(FRHICommandListBase& CmdList);

};

// Engine\Source\Runtime\RHI\Public\RHICommandListCommandExecutes.inl void
FRHICommandDrawPrimitive::Execute(FRHICommandListBase& CmdList) {

        RHISTAT(DrawPrimitive); INTERNAL_DECORATOR(RHIDrawPrimitive)(BaseVertexIndex,
        NumPrimitives,                                                  NumInstances);
}
```

It can be seen that all RHI instructions are pre-declared and implemented. There are nearly 100 types of RHI rendering instructions (as shown below). The rendering thread creates these declared RHI instructions and can be pushed into the RHI thread queue and executed at the appropriate time.

```
FRHICOMMAND_MACRO(FRHICommandUpdateGeometryCacheBuffer)
FRHICOMMAND_MACRO(FRHISubmitFrameToEncoder)
FRHICOMMAND_MACRO(FLocalRHICommand)
FRHICOMMAND_MACRO(FRHISetSpectatorScreenTexture)
FRHICOMMAND_MACRO(FRHISetSpectatorScreenModeTexturePlusEyeLayout)
FRHICOMMAND_MACRO(FRHISyncFrameCommand)
FRHICOMMAND_MACRO(FRHICommandStat)
FRHICOMMAND_MACRO(FRHICommandRHIThreadFence)
FRHICOMMAND_MACRO(FRHIAsyncComputeSubmitList)
FRHICOMMAND_MACRO(FRHICommandWaitForAndSubmitSubListParallel)
FRHICOMMAND_MACRO(FRHICommandWaitForAndSubmitSubList)
FRHICOMMAND_MACRO(FRHICOMMAND_MACRO(FRHICommandWaitForAndSubmitSubListParallel)
CommandWaitForAndSubmitRTSubList) FRHICOMMAND_MACRO(FRHICommandSubmitSubList)
FRHICOMMAND_MACRO(FRHICommandBeginUpdateMultiFrameResource)
FRHICOMMAND_MACRO(FRHICommandEndUpdateMultiFrameResource)
```

```
FRHICOMMAND_MACRO(FRHICommandBeginUpdateMultiFrameUAV)
FRHICOMMAND_MACRO(FRHICommandEndUpdateMultiFrameUAV)
FRHICOMMAND_MACRO(FRHICommandSetGPUMask)
FRHICOMMAND_MACRO(FRHICommandWaitForTemporalEffect)
FRHICOMMAND_MACRO(FRHICommandBroadcastTemporalEffect)
FRHICOMMAND_MACRO(FRHICommandSetStencilRef)
FRHICOMMAND_MACRO(FRHICommandDrawPrimitive)
FRHICOMMAND_MACRO(FRHICommandDrawIndexedPrimitive)
FRHICOMMAND_MACRO(FR                                HICommandSetBlendFactor)
FRHICOMMAND_MACRO(FRHICommandSetStreamSource)
FRHICOMMAND_MACRO(FRHICommandSetViewport)
FRHICOMMAND_MACRO(FRHICommandSetStereoViewport)
FRHICOMMAND_MACRO(FRHICommandSetScissorRect)
FRHICOMMAND_MACRO(FRHICommandSetRenderTargets)
FRHICOMMAND_MACRO(FRHICommandBeginRenderPass)
FRHICOMMAND_MACRO(FRHICommandEndRenderPass)
FRHICOMMAND_MACRO(FRHICommand                                NextSubpass)
FRHICOMMAND_MACRO(FRHICommandBeginParallelRenderPass)
FRHICOMMAND_MACRO(FRHICommandEndParallelRenderPass)
FRHICOMMAND_MACRO(FRHICommandBeginRenderSubPass)
FRHICOMMAND_MACRO(FRHICommandEndRenderSubPass)
FRHICOMMAND_MACRO(FRHICommandBeginComputePass)          FRHICOMMAND
_MACRO(FRHICommandEndComputePass)
FRHICOMMAND_MACRO(FRHICommandBindClearMRTValues)
FRHICOMMAND_MACRO(FRHICommandSetGraphicsPipelineState)
FRHICOMMAND_MACRO(FRHICommandAutomaticCacheFlushAfterComputeShader)
FRHICOMMAND_MACRO(FRHICommandFlushComputeShaderCache)
FRHICOMMAND_MACRO(FRHICommandDrawPrimitiveIndirect)
FRHICOMMAND_MACRO(FRHICommandDrawIndexedIndirect)
FRHICOMMAND_MACRO(FRHICommandDrawIndexedPrimitiveIndirect)
FRHICOMMAND_MACRO(FRHICommandSetDepthBounds)
FRHICOMMAND_MACRO(FRHICommandClearUAVFloat)
FRHICOMMAND_MACRO(FRHICommandClearUAVUint)
FRHICOMMAND_MACRO(FRHICommandCopyToResolveTarget)
FRHICOMMAND_MACRO(FRHICommandCopyTexture)
FRHICOMMAND_MACRO(FRHICommandResummarizeHTile)
FRHICOMMAND_MACRO(FRHICommandTransitionTexturesDepth)
FRHICOMMAND_MACRO(FRHICommandTransitionTextures)
FRHICOMMAND_MACRO(FRHICommandTransitionTexturesArray)
FRHICOMMAND_MACRO(FRHICommandTransitionTexturesPipeline)
FRHICOMMAND_MACRO(FRHICommandTransitionTexturesArrayPipeline)
FRHICOMMAND_MACRO(FRHICommandClearColorTexture)
FRHICOMMAND_MACRO(FRHICommandClearDepthStencilTexture)
FRHICOMMAND_MACRO(FRHICommandClearColorTextures)
FRHICOMMAND_MACRO(FRHICommandSetGlobalUniformBuffers)
FRHICOMMAND_MACRO(FRHICommandBuildLocalUniformBuffer)
FRHICOMMAND_MACRO(FRHICommandBeginRenderQuery)          FRHICOMMAND
_MACRO(FRHICommandEndRenderQuery)
FRHICOMMAND_MACRO(FRHICommandCalibrateTimers)
FRHICOMMAND_MACRO(FRHICommandPollOcclusionQueries)
FRHICOMMAND_MACRO(FRHICommandBeginScene)
FRHICOMMAND_MACRO(FRHICommandEndScene)
FRHICOMMAND_MACRO(FRHICommandBeginFrame)
FRHICOMMAND_MACRO(FRHICommandEndFrame)
FRHICOMMAND_MACRO(FRHICommandBeginDrawingViewport)
FRHICOMMAND_MACRO(FRHICommandEndDrawingViewport)
FRHICOMMAND_MACRO(FRHICommandInvalidateCachedState)     FRHICOMMAND
_MACRO(FRHICommandDiscardRenderTargets)
```

```
FRHICOMMAND_MACRO(FRHICommandDebugBreak)
FRHICOMMAND_MACRO(FRHICommandUpdateTextureReference)
FRHICOMMAND_MACRO(FRHICommandUpdateRHIResources)
FRHICOMMAND_MACRO(FRHICommandCopyBufferRegion)
FRHICOMMAND_MACRO(FRHICommandCopyBufferRegions)
FRHICOMMAND_MACRO(FRHICommandCl earRayTracingBindings)
FRHICOMMAND_MACRO(FRHICommandRayTraceOcclusion)
FRHICOMMAND_MACRO(FRHICommandRayTraceIntersection)
FRHICOMMAND_MACRO(FRHICommandRayTraceDispatch)
FRHICOMMAND_MACRO(FRHICommandSetRayTracingBindings)
FRHICOMMAND_MACRO(FClearCachedRenderingDataCommand)
FRHICOMMAND_MACRO(FClearCachedElementDataCommand)
```

2.5.4 Interaction between game thread and rendering thread

This section will describe the data exchange mechanism and implementation details between threads. First, let's look at how the game thread passes data to the rendering thread.

When the game thread is ticking, it will enter the call of the rendering module through objects such as UGameEngine, FViewport, and UGameViewportClient:

```
void UGameEngine::Tick(floatDeltaSeconds,boolbIdleMode ) {

    UGameEngine::RedrawViewports() {

        void FViewport::Draw(boolbShouldPresent) {

            void UGameViewportClient::Draw() {

                //calculateViewFamily,ViewVarious attributes of
                ULocalPlayer::CalcSceneView(); //Sending
                rendering commands
                FRendererModule::BeginRenderingViewFamily() {

                    World->SendAllEndOfFrameUpdates(); //Creating
                    a scene renderer
                    FSceneRenderer* SceneRenderer =
FSceneRenderer::CreateSceneRenderer(ViewFamily, ...);
                    //Sends scene drawing instructions to the rendering thread.
                    ENQUEUE_RENDER_COMMAND(FDrawSceneCommand)
                    ( [SceneRenderer](FRHICommandListImmediate& {       RHICmdList)

                        RenderViewFamily_RenderThread(RHICmdList, {          SceneRenderer)

                            (......)
                            //Call the drawing interface of the scene renderer.
                            SceneRenderer->Render(RHICmdList); (......)


                        }
                        FlushPendingDeleteRHIResources_RenderThread();
                    });
                }
}}}}
```

As mentioned in the previous chapter, the rendering thread uses objects such as SceneProxy and SceneInfo. So how does the Actor component of the game connect to the data of the scene proxy? And how does it update the data?

First, let's figure out how the game component passes data to SceneProxy. The answer lies in FScene::AddPrimitive:

```
// Engine\Source\Runtime\Renderer\Private\RendererScene.cpp

void FScene::AddPrimitive(UPrimitiveComponent* Primitive) {

	(……)

	//Create a scene proxy for the primitive
	FPrimitiveSceneProxy* PrimitiveSceneProxy = Primitive->CreateSceneProxy(); Primitive->SceneProxy
	= PrimitiveSceneProxy;
	if(!PrimitiveSceneProxy) {

		return;
	}

	//Create scene information for primitive scene proxy
	FPrimitiveSceneInfo* PrimitiveSceneInfo = new FPrimitiveSceneInfo(Primitive, this); PrimitiveSceneProxy-
	>PrimitiveSceneInfo = PrimitiveSceneInfo;

	(……)

	FScene* Scene = this;

	ENQUEUE_RENDER_COMMAND(AddPrimitiveCommand)(
		[Params = MoveTemp(Params), Scene, PrimitiveSceneInfo, PreviousTransform =
MoveTemp(PreviousTransform)](FRHICommandListImmediate& RHICmdList)
		{
			FPrimitiveSceneProxy* SceneProxy = Params.PrimitiveSceneProxy;

			(……)

			SceneProxy->CreateRenderThreadResources(); //In the rendering threadSceneInfo
			Add to the scene. Scene-
			>AddPrimitiveSceneInfo_RenderThread(PrimitiveSceneInfo,
PreviousTransform);
		});
}
```

The key sentence abovePrimitive->CreateSceneProxy()is to create the PrimitiveSceneProxy corresponding to the component. In the constructor of PrimitiveSceneProxy, all the data of the component is copied:

```
FPrimitiveSceneProxy::FPrimitiveSceneProxy(constUPrimitiveComponent* InComponent, FName InResourceName)

:
	CustomPrimitiveData(InComponent->GetCustomPrimitiveData())
,	TranslucencySortPriority(FMath::Clamp(InComponent->TranslucencySortPriority,
SHRT_MIN, SHRT_MAX))
```

```
,       Mobility(InComponent->Mobility)
,       LightmapType(InComponent->LightmapType)
,       StatId()
,       DrawInGame(InComponent->IsVisible())
,       DrawInEditor(InComponent->GetVisibleFlag())
,       bReceivesDecals(InComponent->bReceivesDecals)
,

(......)

{
    (......)
}
```

After copying the data, the game thread modifies the data of PrimitiveComponent, while the rendering thread modifies or accesses the data of PrimitiveSceneProxy, which do not interfere with each other, avoid synchronization of critical sections and locks, and ensure thread safety.
However, there is still a question here, that is, a copy of the data will be copied when creating PrimitiveSceneProxy, but after the creation, how does PrimitiveComponent update the data to PrimitiveSceneProxy?

It turns out that ActorComponent has several tags. As long as these tags are markedtrue,the update interface will be called at the appropriate time to get the update:

```
// Engine\Source\Runtime\Engine\Classes\Components\ActorComponent.h

classENGINE_API UActorComponent: public UObject, public IInterface_AssetUserData {

protected:
    //The following interfaces update the corresponding status respectively, and subclasses can rewrite
    them to implement their own update logic. virtualvoid DoDeferredRenderUpdates_Concurrent() {

        (......)

        if(bRenderStateDirty)
        {
            RecreateRenderState_Concurrent();
        }
        else
        {
            if(bRenderTransformDirty) {

                SendRenderTransform_Concurrent();
            }
            if(bRenderDynamicDataDirty) {

                SendRenderDynamicData_Concurrent();
            }
        }
    }
    virtualvoid CreateRenderState_Concurrent(FRegisterComponentContext* Context) {

        bRenderStateCreated =true;

        bRenderStateDirty =false;
```

```
        bRenderTransformDirty =         false;
        bRenderDynamicDataDirty         =false;
    }
    virtual     void SendRenderTransform_Concurrent()
    {
        bRenderTransformDirty       =false;
    }
    virtualvoid SendRenderDynamicData_Concurrent() {

        bRenderDynamicDataDirty =false;
    }

private:
    uint8 bRenderStateDirty:1;//Whether the component's rendering state is dirty uint8
    bRenderTransformDirty:1;//Whether the component's transformation matrix is dirty uint8
    bRenderDynamicDataDirty:1;//Is the component's rendering dynamic data dirty?
};
```

The protected interface above is used to refresh the component data to the corresponding SceneProxy. Specific component subclasses can rewrite it to customize their own update logic. For example, the ULightComponenttransformation matrix update logic is as follows:

```
// Engine\Source\Runtime\Engine\Private\Components\LightComponent.cpp

void ULightComponent::SendRenderTransform_Concurrent() {

    //Updates the transformation information to the scene.
    GetWorld()->Scene->UpdateLightTransform(this);
    Super::SendRenderTransform_Concurrent();
}
```

The sceneUpdateLightTransformwill assemble the component data and send the data to the rendering thread for execution:

```
// Engine\Source\Runtime\Renderer\Private\RendererScene.cpp

void FScene::UpdateLightTransform(ULightComponent* Light) {

    if(Light->SceneProxy)
    {
        //Assemble component data into a structure (note that you cannotComponentThe address is passed to the rendering thread, but all the data to be updated is copied
1 serving)
        FUpdateLightTransformParameters         Parameters;
        Parameters.LightToWorld         = Light->GetComponentTransform().ToMatrixNoScale(); Light-
        Parameters.Position =           >GetLightPosition();
        FScene* Scene = this;
        FLightSceneInfo* LightSceneInfo = Light->SceneProxy->GetLightSceneInfo(); //Send data to the
        rendering thread for execution.
        ENQUEUE_RENDER_COMMAND(UpdateLightTransform)(
            [Scene, LightSceneInfo, Parameters](FRHICommandListImmediate& RHICmdList) {

                FScopeCycleCounter Context(LightSceneInfo->Proxy->GetStatId()); //Perform data updates
                in the rendering thread.
                Scene->UpdateLightTransform_RenderThread(LightSceneInfo, Parameters);
            });
```

```
    }
}

void FScene::UpdateLightTransform_RenderThread(FLightSceneInfo* LightSceneInfo,const
FUpdateLightTransformParameters& Parameters) {

    (……)

    //Update the transformation matrix.
    LightSceneInfo->Proxy->SetTransform(Parameters.LightToWorld, Parameters.Position);

    (……)
}
```

At this point, the logic of how components update data to the scene proxy is finally clear.

It is important to note that some interfaces such as FScene and FSceneProxy are called in the game thread, while some interfaces (generally with the_RenderThreadsuffix) are called in the rendering thread. Remember not to call across threads, otherwise it will cause competition conditions and even cause the program to crash.

2.5.5 Synchronization of game thread and rendering thread

As mentioned earlier, the game thread cannot be ahead of the rendering thread by more than one frame, otherwise the game thread will wait for the rendering thread to finish processing. Their synchronization mechanism involves two key concepts:

```
// Engine\Source\Runtime\RenderCore\Public\RenderCommandFence.h

//Render Command Fence
classRENDERCORE_API        FRenderCommandFence
{
public:
    //Adds a fence to the rendering command queue.bSyncToRHIAndGPUIs it synchronized?RHIandGPUexchangeBuffer,Otherwise just wait
    for the rendering thread. void BeginFence(boolbSyncToRHIAndGPU=false);

    //Wait for the fence to be executed.bProcessGameThreadTasksNo effect.
    void Wait(boolbProcessGameThreadTasks =false)const;

    //Whether the fence is completed.
    bool IsFenceComplete()const;

private:
    mutable FGraphEventRef CompletionEvent;//Handle the synchronization completion event
    ENamedThreads::Type TriggerThreadIndex;//The type of thread that needs to be triggered after processing.
};

// Engine\Source\Runtime\Engine\Public\UnrealEngine.h class
FFrameEndSync
{
    FRenderCommandFence Fence[2];//Renders a fence pair.
    int32 EventIndex;//Current Event Index public:
```

```
    //Synchronize the game thread and rendering thread.bAllowOneFrameThreadLagWhether to allow the rendering thread to be
    delayed by one frame. void Sync(boolbAllowOneFrameThreadLag ) {

        Fence[EventIndex].BeginFence(true);//Open fence, force synchronizationRHIandGPUExchange chain.

        boolbEmptyGameThreadTasks = !
        FTaskGraphInterface::Get().IsThreadProcessingTasks(ENamedThreads::GameThread);

        //    Ensure that the game thread runs the task at least once.
        if    (bEmptyGameThreadTasks)
        {
            FTaskGraphInterface::Get().ProcessThreadUntilIdle(ENamedThreads::GameThread);
        }

        //If delay is allowed, swap event indices.
        if( bAllowOneFrameThreadLag ) {

            EventIndex = (EventIndex +1) %2;
        }

        (......)

        //Open the fence and wait.
        Fence[EventIndex].Wait(bEmptyGameThreadTasks);
    }
};
```

The usage`FFrameEndSync`is in`FEngineLoop::Tick`:

```
// Engine\Source\Runtime\Launch\Private\LaunchEngineLoop.cpp

void FEngineLoop::Tick() {

    (......)

    //Adds synchronization events for the game thread and the rendering thread at the end of the frame of the engine loop. {

        staticFFrameEndSync FrameEndSync;//Local static variables, thread safe. staticauto
        CVarAllowOneFrameThreadLag = IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT(
"r.OneFrameThreadLag"));
        //Synchronizes the game and rendering threads, allowing a frame of delay to be controlled by console commands. Enabled by
        default. FrameEndSync.Sync( CVarAllowOneFrameThreadLag->GetValueOnGameThread() !=0);
    }

    (......)
}
```

2.6 Conclusion on Multithreaded Rendering

Parallel computing architecture has become the standard for modern engines. UE's multi-threaded rendering is an inevitable product with the birth of multi-core CPUs and a new generation of graphics APIs. But for now, the rendering thread is often still a single thread (although it can be

partially parallelized with the help of TaskGraph). Ideally, multiple rendering threads generate rendering commands in parallel and independently, and do not need a main thread to drive them. Any thread can be used as a working thread (that is, there is no named thread for UE), and any thread can initiate computing tasks to avoid functional threads at the operating system level. This requires the continuous evolution of the operating system, graphics API, and computer language to achieve this.

The recently released UE4.26 has already popularized RDG, which can automatically crop and optimize rendering passes and resources, and is a powerful tool for improving the overall parallel processing of the engine.

This article was originally expected to be completed in about 2 months, but it actually took more than 3 months, almost exhausting all the author's spare time. There were originally many technical chapters that needed to be added, but the length and time were exceeded, so I had to give up. I hope this series of articles will be helpful to readers who are learning UE. Thank you for your attention and collection.

References

- Unreal Engine 4 Sources Unreal
- Engine 4 Documentation
- Rendering and Graphics
- Materials
- Graphics Programming
- Unreal Engine 4 Rendering
- Threaded Rendering
- "The Elephant is Invisible-A Brief Analysis of Unreal Engine Programming"

- Fang Yanliang - Analysis of Unreal 4 Rendering System Architecture UE4
- Render System Sheet
- Inside UE4
- Exploring in UE4
- Game Engine Architecture
- C++ reference
- Object-oriented multithreaded programming in C++
- Win32 Multithreaded Programming
- Windows Core Programming (5th Edition)
- "Effective C++"
- "C++ Concurrency In Action" "Modern
- Concurrency in Depth" "Multithreaded
- Programming Guide" Windows 95
-
- Multi-core processors
- Amdahl's law
- Thread-local storage
- Covering Multithreading Basics Multi-
- Threaded Programming: C++11
- In-depth understanding of Coroutine and its principles and Coroutine in Kotlin C+
- +11 Multithreading
- C++ portability and cross-platform development[6]: multithreading
- volatile (computer programming)
- Talk about volatile in C/C++
- Practical Parallel Rendering with DirectX 9 and 10
- Multi-engine synchronization
- Understanding DirectX Multithreaded Rendering Performance by Experiments Performance,
- Methods, and Practices of DirectX 11 Multithreaded Rendering Introduction to Multithreaded
- rendering and the usage of Deferred Contexts in DirectX 11 Practical Parallel Rendering with
- DirectX 9 and 10
- Comparison of D3D11 and D3D12 multithreaded rendering frameworks
- Render graphs and Vulkan — a deep dive
- DirectX 12: Performance Comparison Between Single- and Multithreaded Rendering when
- Culling Multiple Lights
- Vulkan - High Performance Rendering
- Vulkan multi-threaded rendering
- Vulkan Multi-Threading
- The impact of the new generation graphics API VULKAN on AR and VR Fork-
- Join Model
- Fiber (computer science)
- Evaluation of multi-threading in Vulkan
-

- Metal: Command Organization and Execution Model Task-
- based Multithreading - How to Program for 100 cores Multi-
- Threaded Game Engine Design
- Game Engine Essay 0x04: Parallel Computing Architecture
- Multithreaded Rendering & Graphics Jobs
- Optimizing Graphics in Unity
- Frostbite Rendering Architecture and Real-time Procedural Shading & Texturing Techniques (GDC 2007)
- FrameGraph: Extensible Rendering Architecture in Frostbite
- Parallelizing the Naughty Dog engine using fibers Destiny's
- Multi-threaded Renderer Architecture talk Multithreading the
- Entire Destiny Engine
- Everything You Need to Know About Multithreaded Rendering in Fortnite bgfx
-
- Unreal 4 Task Graph System Introduction
- Simple analysis of TaskGraph system in UE4
- Directed acyclic graph
- Scalability for All: Unreal Engine* 4 with Intel UE
- Advanced Performance Analysis Technology
- UE4 main loop information
- Intel® ISPC User's Guide
- Multithreaded rendering
- Rendering thread in UE4
- UE4 main thread and rendering thread synchronization
- Bringing Fortnite to Mobile with Vulkan and OpenGL ES

https://github.com/pe7yu