Voices from Community – Certificate

We'll set three grading levels for **Certificates**

- **Graduated**: submitted all assignments in time with grading over 60 points
- **Excellent**: Passed and with 2 assignments achieving 100 points (HW1 not counted)
- **Outstanding**: Passed and with 3 assignments achieving 100 points (HW1 not counted)

Rewarding for Graduation:

- Piccolo T-Shirt

Submission Window: September 1st 2022 to October 31st 2022

Voices from Community – WHAT'S THE NEXT

Future plan for Piccolo Community

- Code explained
  - on 10th October
  - details will be announced in our WeChat groups
- Various technical content
- Community activities
- releases of Piccolo versions

关注公众号，回复「入群」，加入Piccolo微信群

You can also send your fabulous advices about code explained via e-mail:

piccolo-gameengine@boomingtech.com

Q&A

- Q1: Many games are ruined by cheaters. Could we get rid of cheaters once and for all?

- Q2: What's the difference between micro-service and distributed server architecture?

- Q3: We're always wishing we could battle with friends around the world.
  How could we implement a global server?

Lecture 20

Data-Oriented Programming and Job System

Advanced Topics

WANG XI

GAMES 104

2022

Code Execution Is Not As Simple As It Looks

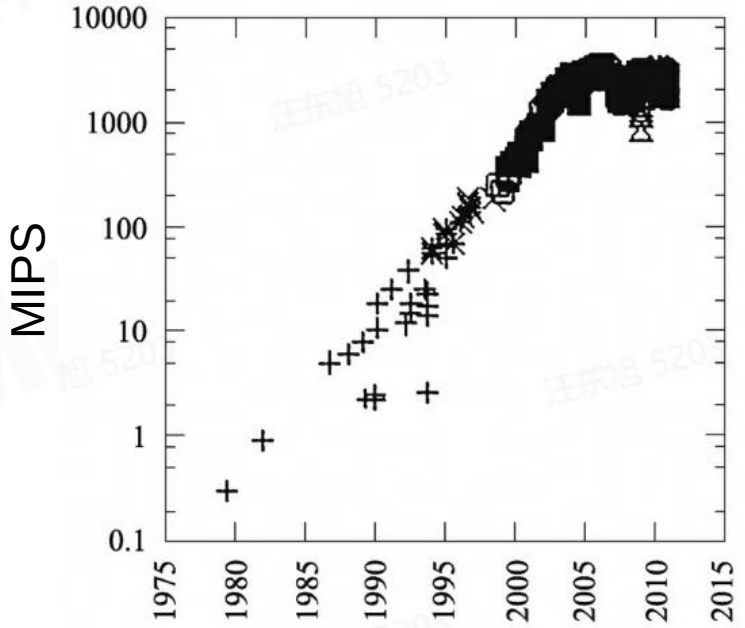
Code is executed on top of specific hardware and operating system

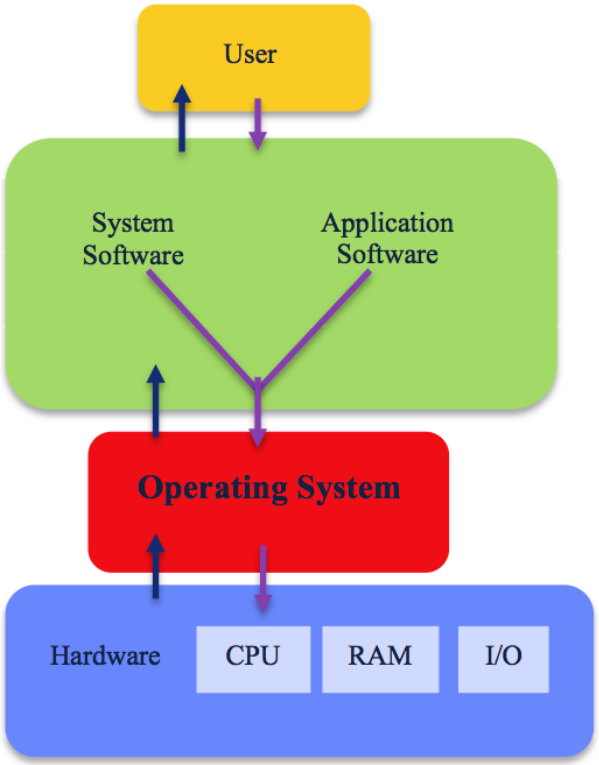
- Hardware and OS must be considered if we want to write a high performance program

Basics of Parallel Programming

Ceiling of Moore's Law and Multi-Cores

- The number of transistors in a dense integrated circuit (IC) **doubles about every two years**
- In these years, chip densities are **no longer** doubling every two years
- Multi-core processor becomes the new industry trend





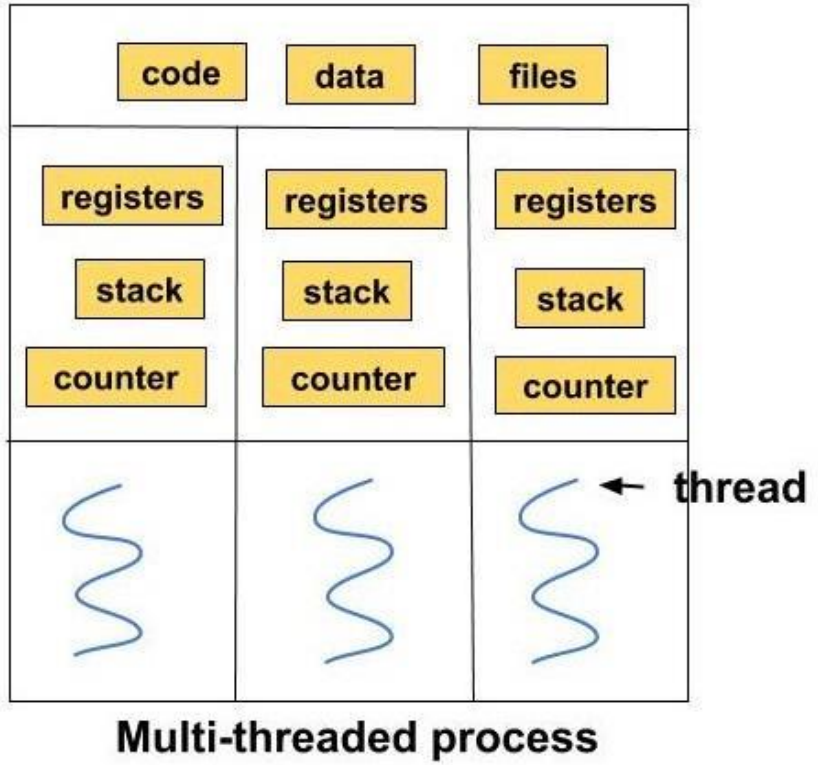
Process and Thread

Process

- The instance of an application (or program)
- Has its own individual region of memory

Thread

- ***Preemptive multitasking***
- The smallest unit of task that can be scheduled by OS
- Must reside in a process
- Threads in the same process share the same region of memory

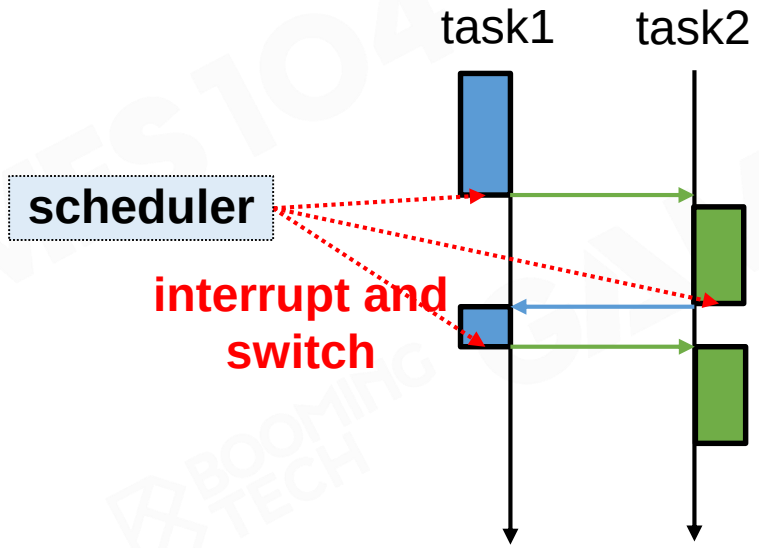


Multi-threaded process

Types of Multitasking

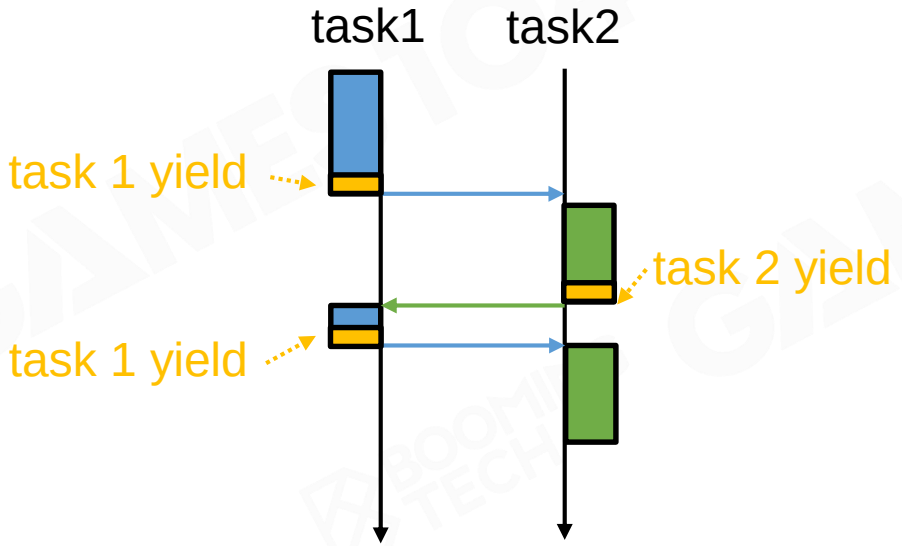
Preemptive Multitasking

- Currently executing task **can be interrupted at a time** decided by the scheduler
- Scheduler determines which task to be executed next
- Applied in most operating systems



Non-preemptive Multitasking

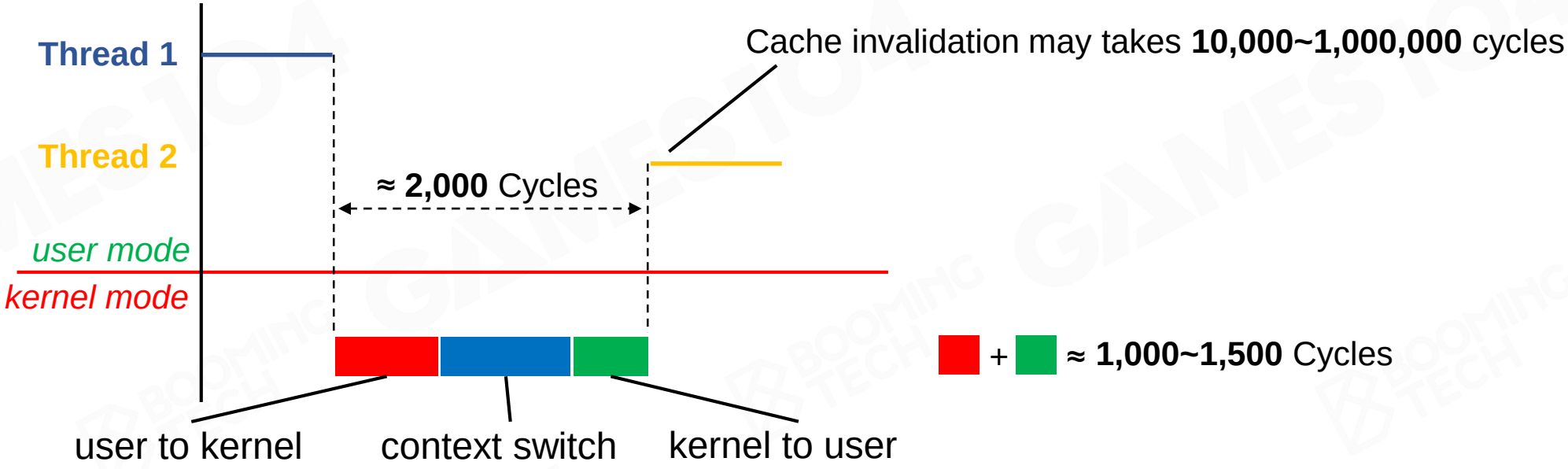
- Tasks must be **explicitly programmed to yield control**
- Tasks must cooperate for the scheduling scheme to work
- Currently many real-time operating systems (RTOS) also support this kind of scheduling



Thread Context Switch

Store the state of a thread and resume the execution at a later point

- State including registers, stack and other OS required data
- Thread context switch implies extra user-kernel mode switch
- Cache invalidation after context switch has even more cost



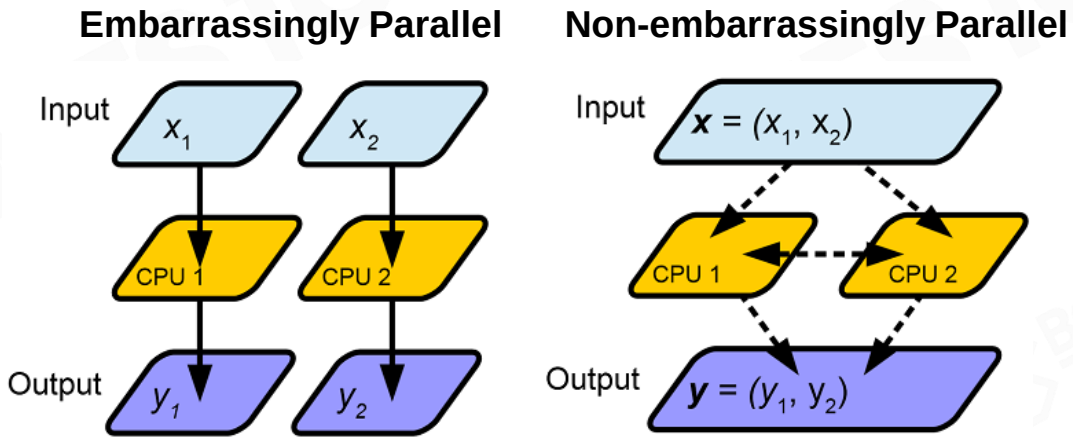
Parallel Problems in Parallel Computing

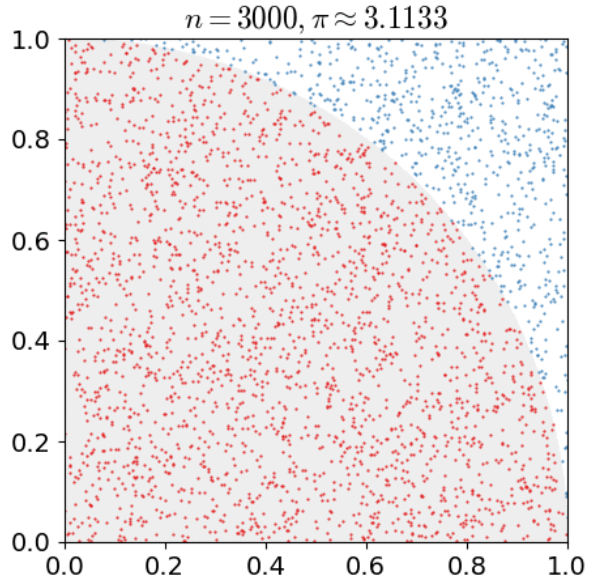
Embarrassingly Parallel Problem (or Perfectly Parallel)

- Little or no dependency or need for communication between parallel tasks

Non-embarrassingly Parallel Problem

- Communication is needed between parallel tasks





Monte Carlo algorithm is a typical example of embarrassingly parallel

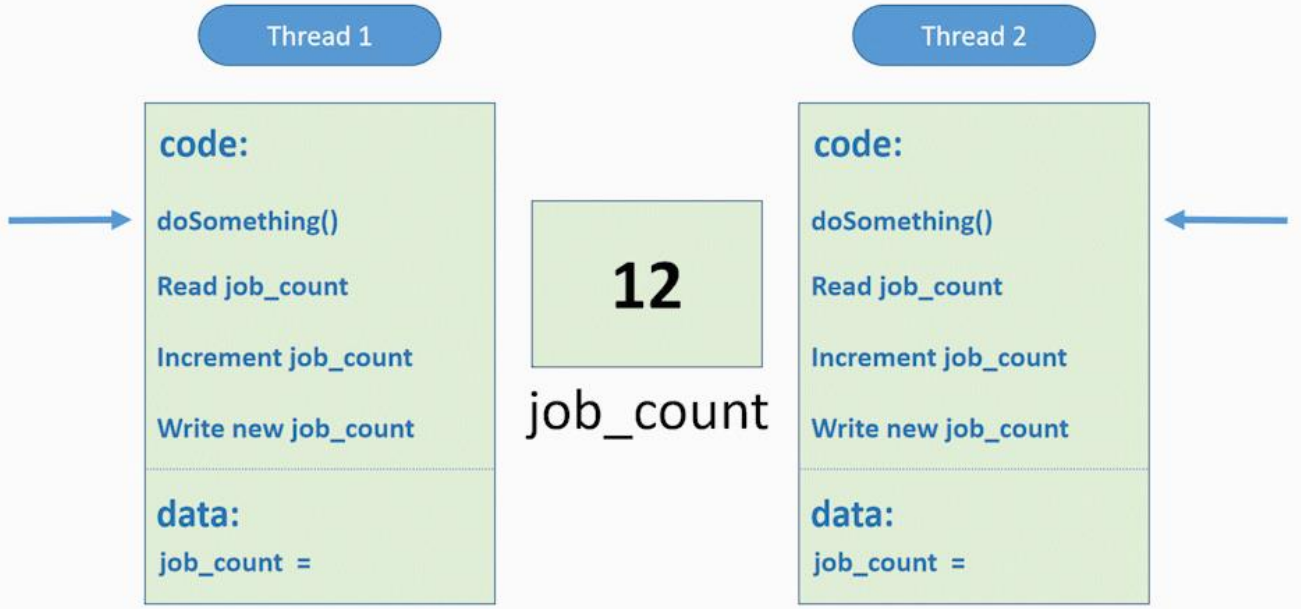
Data Race in Parallel Programming

Multiple threads in a single process access the same memory location concurrently

- At least one of the accesses is for writing

```
while(job_count < 100)
{
    doSomething();
    job_count++;
}
```

- Read `job_count`
- Compute `job_count + 1`
- Write **new** `job_count`

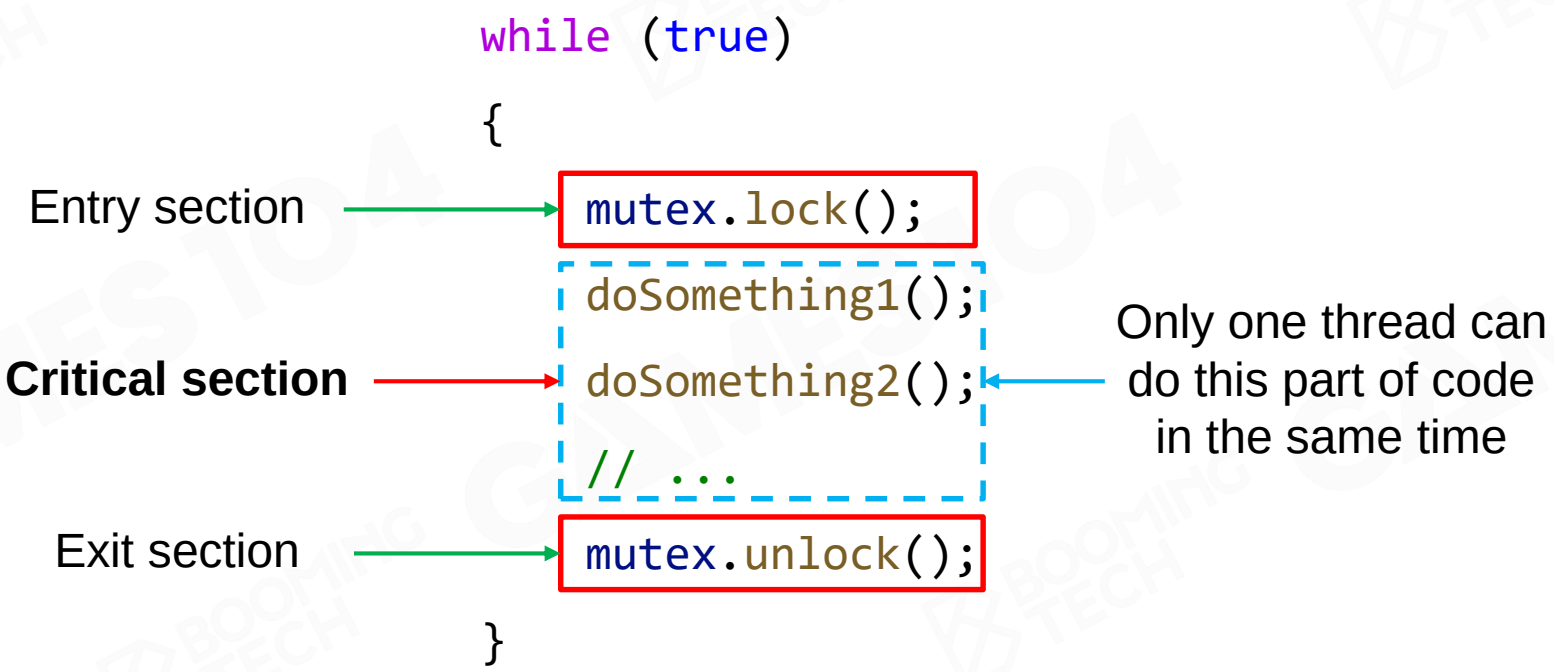


2 thread expect job_count + 2, but actually +1

Blocking Algorithm - Locking Primitives

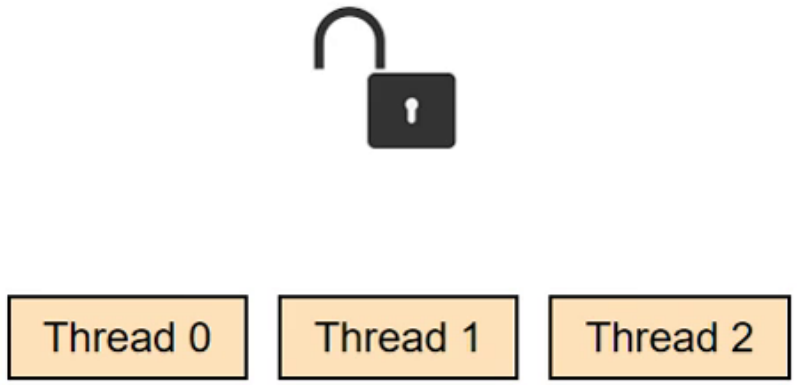
Lock

- Only one thread can acquire the lock at a time
- Make a ***critical section*** for shared resource access

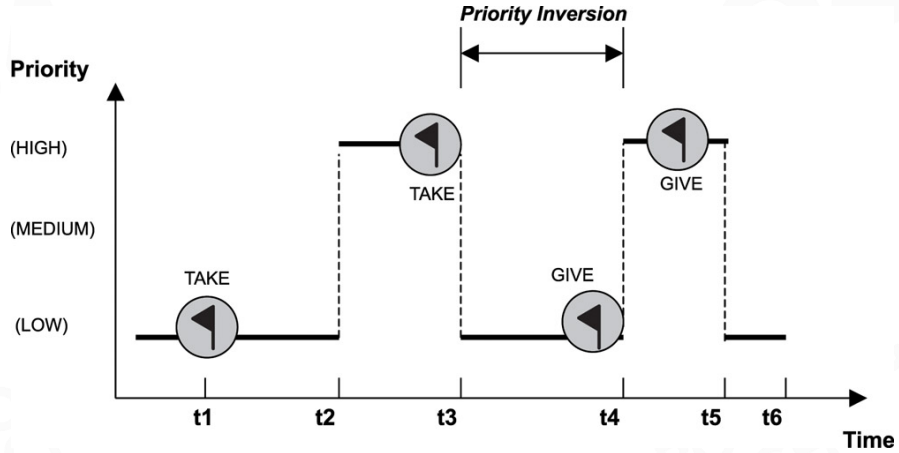


Other Issues with Locks

- Thread suspending and resuming will bring performance overhead
- Suspending threads never get resumed if the thread that acquires the lock exits abnormally
- Priority Inversion
  - A higher priority task attempts to acquire the lock that is already acquired by a lower priority task



Deadlock Caused by Thread Crash



Priority Inversion

Atomic Operation : Lock-free Programming

Atomic Loads and Stores

- **Load**: Load data from shared memory to either a register or thread-specific memory
- **Store**: Move data into shared memory

Atomic Read-Modify-Write (RMW)

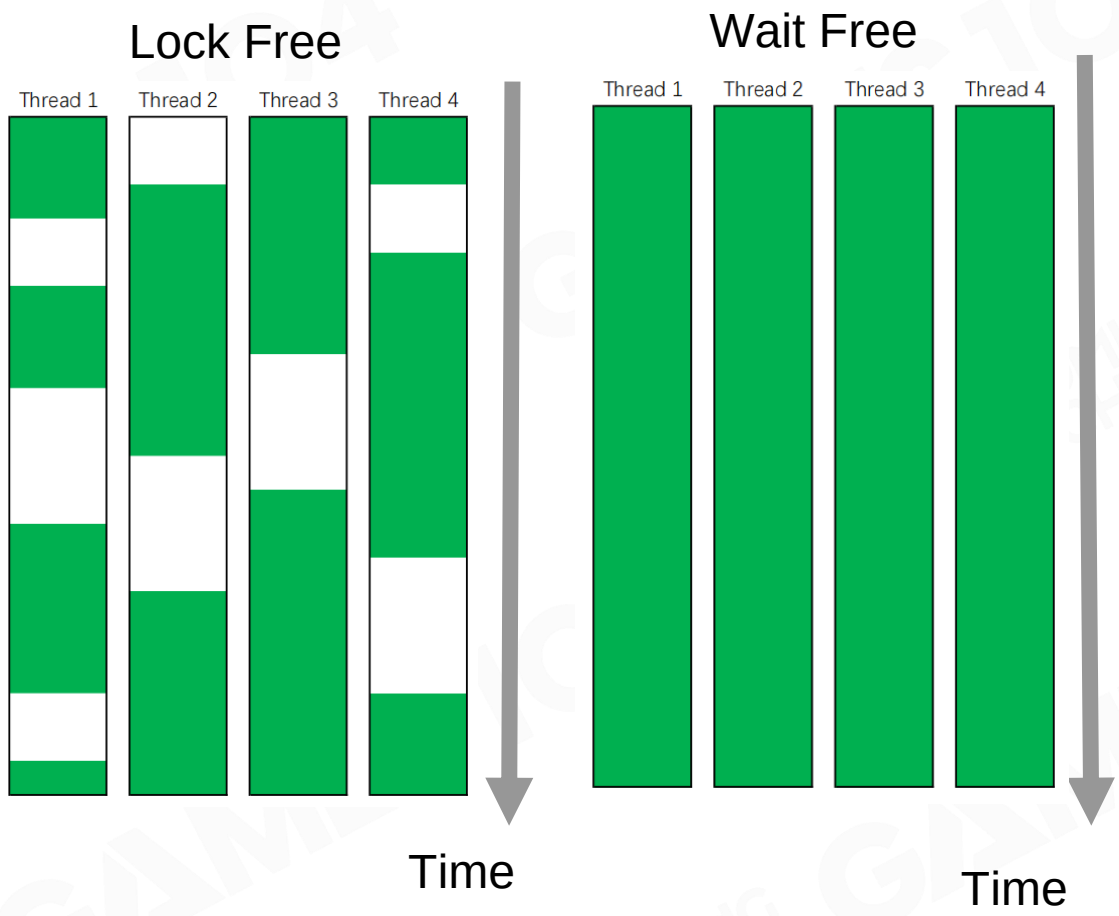
- **Test and Set**: Set 1 to shared memory and return the previous value
- **Compare and Swap (CAS)**: Update the data in shared memory if it equals an expected value
- **Fetch and Add**: Add a value to the data in shared memory and return the previous value
- ...

Operations on atomic types

`atomic_is_lock_free` (C++11)	checks if the atomic type's operations are lock-free (function template)
`atomic_store` (C++11) `atomic_store_explicit` (C++11)	atomically replaces the value of the atomic object with a non-atomic argument (function template)
`atomic_load` (C++11) `atomic_load_explicit` (C++11)	atomically obtains the value stored in an atomic object (function template)
`atomic_exchange` (C++11) `atomic_exchange_explicit` (C++11)	atomically replaces the value of the atomic object with non-atomic argument and returns the old value of the atomic (function template)
`atomic_compare_exchange_weak` (C++11) `atomic_compare_exchange_weak_explicit` (C++11) `atomic_compare_exchange_strong` (C++11) `atomic_compare_exchange_strong_explicit` (C++11)	atomically compares the value of the atomic object with non-atomic argument and performs atomic exchange if equal or atomic load if not (function template)
`atomic_fetch_add` (C++11) `atomic_fetch_add_explicit` (C++11)	adds a non-atomic value to an atomic object and obtains the previous value of the atomic (function template)
`atomic_fetch_sub` (C++11) `atomic_fetch_sub_explicit` (C++11)	subtracts a non-atomic value from an atomic object and obtains the previous value of the atomic (function template)
`atomic_fetch_and` (C++11) `atomic_fetch_and_explicit` (C++11)	replaces the atomic object with the result of bitwise AND with a non-atomic argument and obtains the previous value of the atomic (function template)
`atomic_fetch_or` (C++11) `atomic_fetch_or_explicit` (C++11)	replaces the atomic object with the result of bitwise OR with a non-atomic argument and obtains the previous value of the atomic (function template)
`atomic_fetch_xor` (C++11) `atomic_fetch_xor_explicit` (C++11)	replaces the atomic object with the result of bitwise XOR with a non-atomic argument and obtains the previous value of the atomic (function template)
`atomic_wait` (C++20) `atomic_wait_explicit` (C++20)	blocks the thread until notified and the atomic value changes (function template)
`atomic_notify_one` (C++20)	notifies a thread blocked in atomic_wait (function template)
`atomic_notify_all` (C++20)	notifies all threads blocked in atomic_wait (function template)

Parts of C++ atomic operations library

Lock Free vs. Wait Free



Compiler Reordering Optimizations

```
void function1()
{
    a = b + 1;
    b = 0;
}
```

Turn off optimization:

```
movl    b(%rip), %eax
addl    $1, %eax
movl    %eax, a(%rip)
movl    $0, b(%rip)
```

Same as:

```
void function1()
{
    a = b + 1;
    b = 0;
}
```

Same order

Turn on optimization:

```
movl    b(%rip), %eax
movl    $0, b(%rip)
addl    $1, %eax
movl    %eax, a(%rip)
```

Looks like:

```
void function1()
{
    int temp = b;
    b = 0;
    a = temp + 1;
}
```

Reordered!

Problem of Memory Reordering

- Compilers and CPUs often modify the execution order of instructions to optimize performance
- It's the hard part of parallel programming

Initialize:

```
int a = 0;
int b = 1;
```

Thread 1

```
void function1()
{
    a = b + 1;
    b = 0;
}
```

Thread 2

```
void function2()
{
    while(b!=0)
    {
        continue;
    }
    assert(a==2);
}
```

Will the assertion fire?

Out-of-order Execution by CPUs

For different CPU

- The optimization strategy are significantly different
- Provides different types of memory order guarantees
- Parallel programs require **different processing**

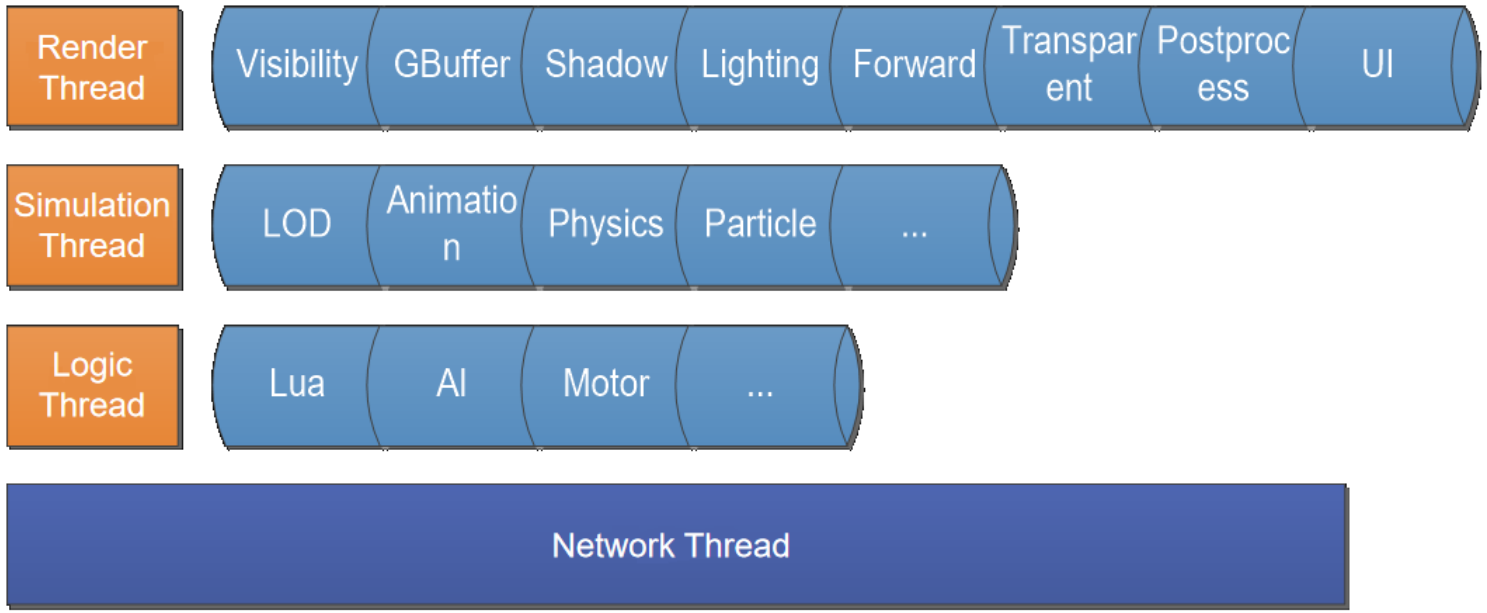
Type	Alpha	ARMv7	MIPS	RISC-V		PA-RISC	POWER	SPARC			x86 [a]	AMD64	IA-64	z/Architecture
				WMO	TSO			RMO	PSO	TSO				
Loads can be reordered after loads	Y	Y	depend on implementation	Y		Y	Y	Y					Y	
Loads can be reordered after stores	Y	Y		Y		Y	Y	Y					Y	
Stores can be reordered after stores	Y	Y		Y		Y	Y	Y	Y				Y	
Stores can be reordered after loads	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic can be reordered with loads	Y	Y		Y			Y	Y					Y	
Atomic can be reordered with stores	Y	Y		Y			Y	Y	Y				Y	
Dependent loads can be reordered	Y													
Incoherent instruction cache/pipeline	Y	Y		Y	Y		Y	Y	Y	Y	Y		Y	

Parallel Framework of Game Engine

Fixed Multi-thread

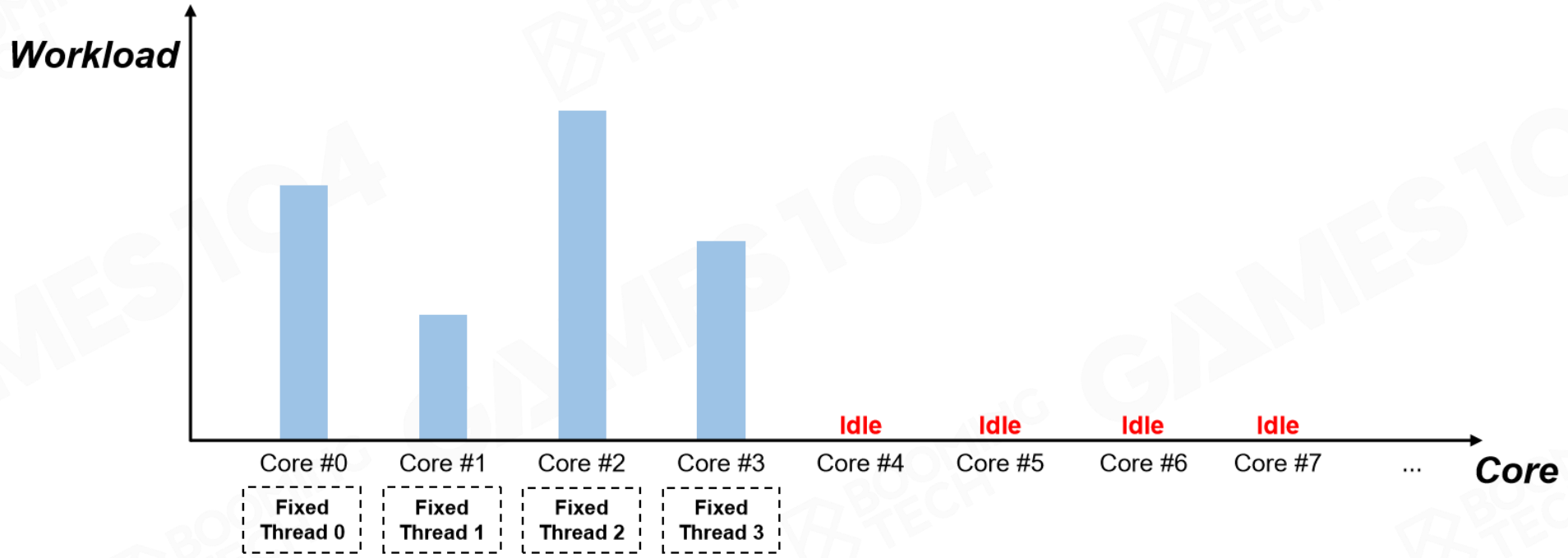
One fixed thread for each part of the game engine logic

- Render, Simulation, Logic, Network, and etc.
- Easy to implement



Issues with Fixed Multi-thread

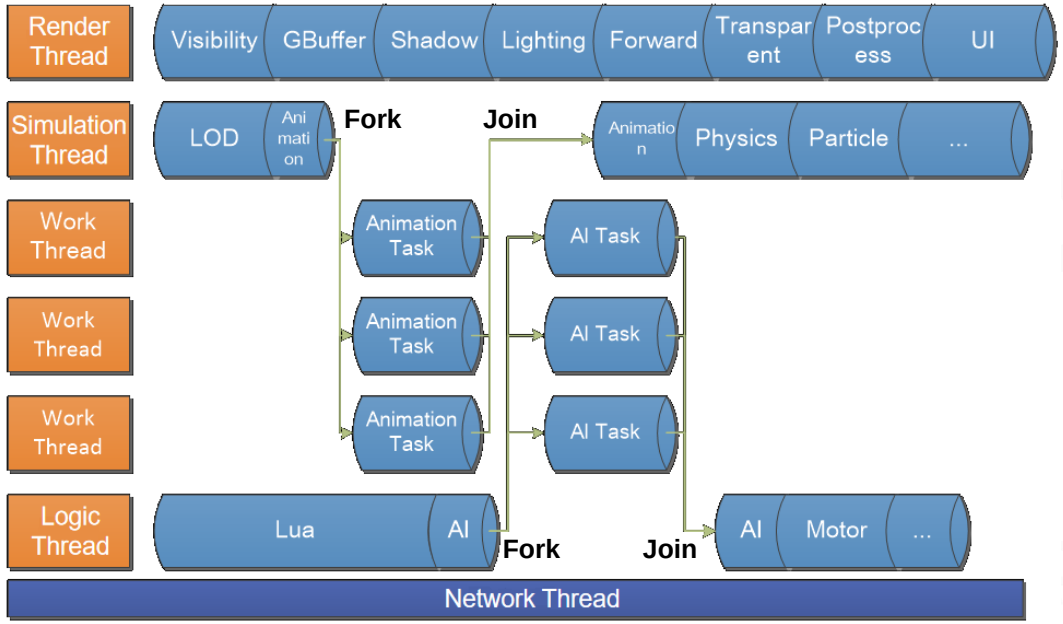
- The workload is unbalanced among threads (cores)
- Unscalable while there are more processor cores

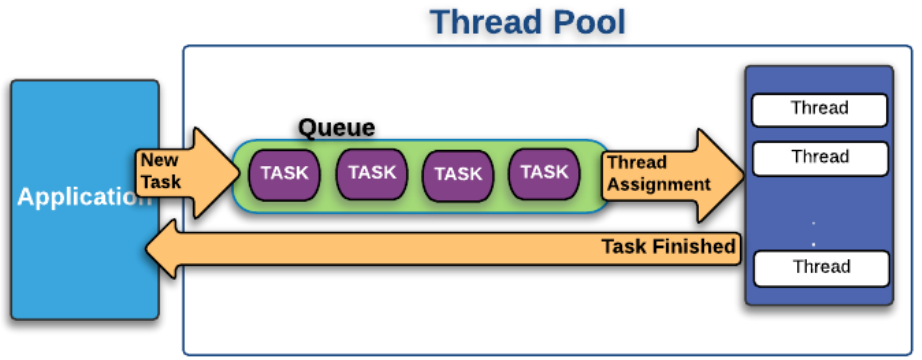


Thread Fork-Join

Fork-join for data-parallelizable work (based on fixed multi-thread)

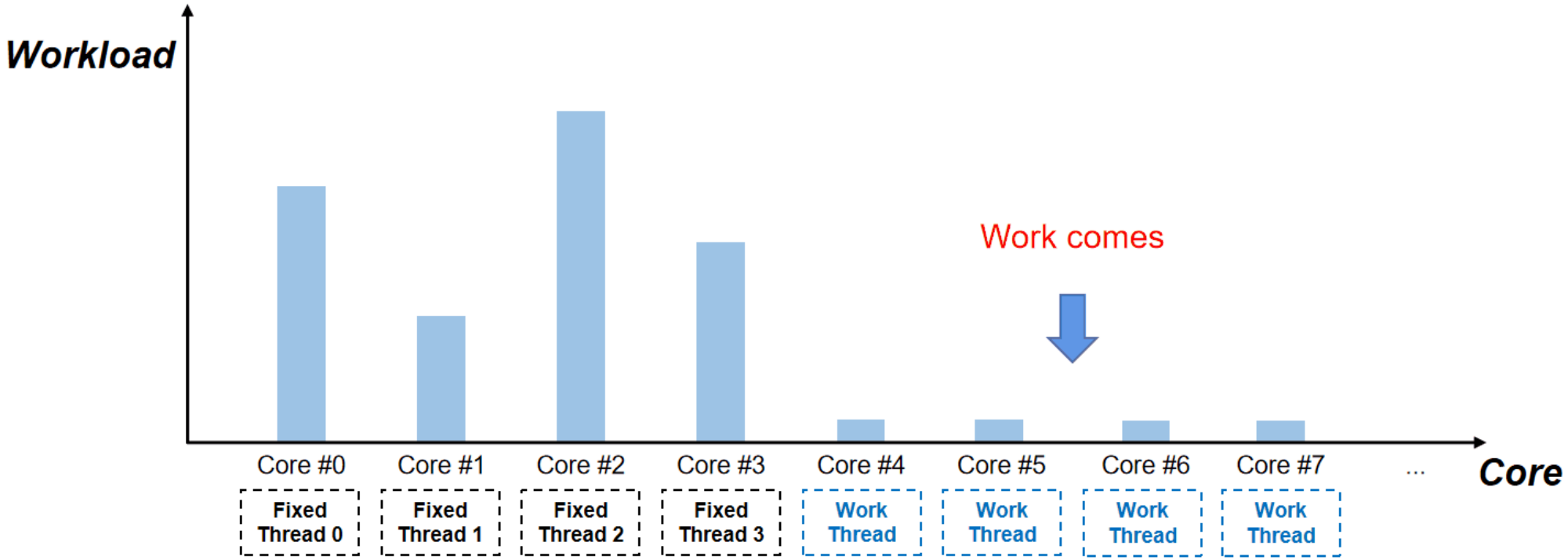
- Use a ***thread pool*** to prevent frequent thread creation/destruction





Problems with Thread Fork-Join

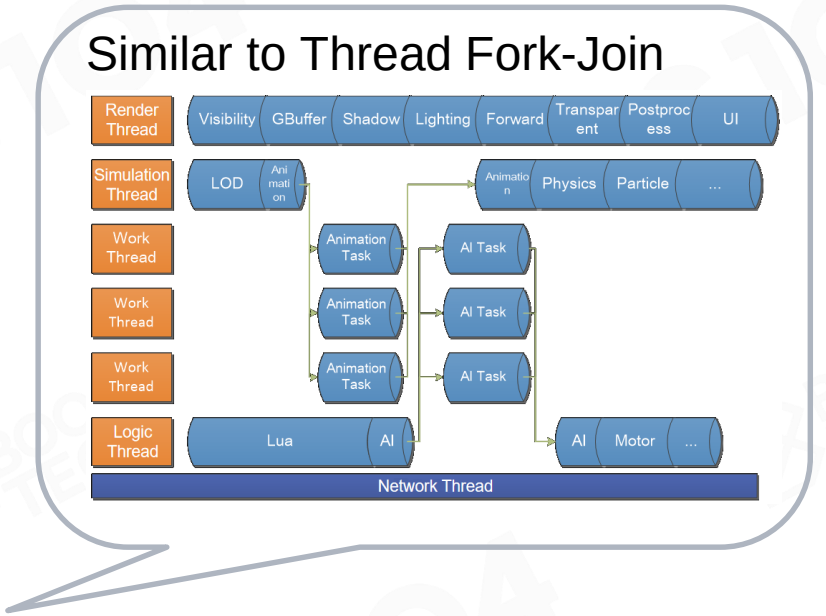
- Not easy for logic programmers (work split, work threads count)
- Too many threads can bring performance overhead on context switch
- The workload is still unbalanced among threads (cores)

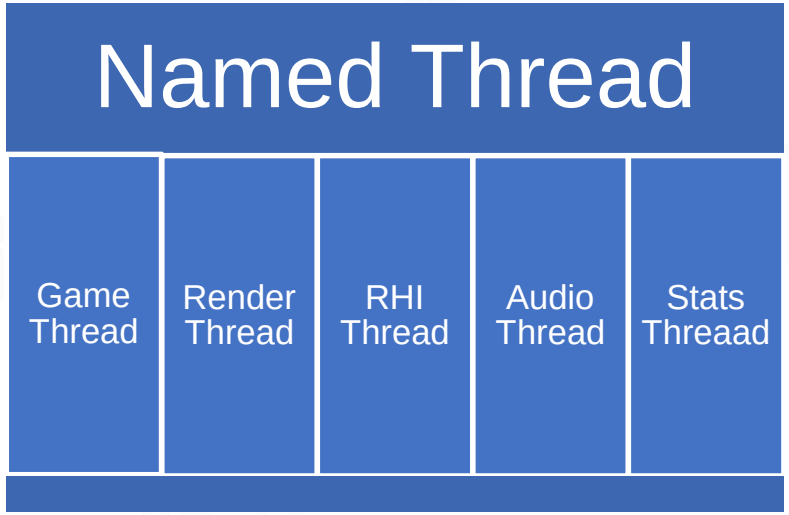


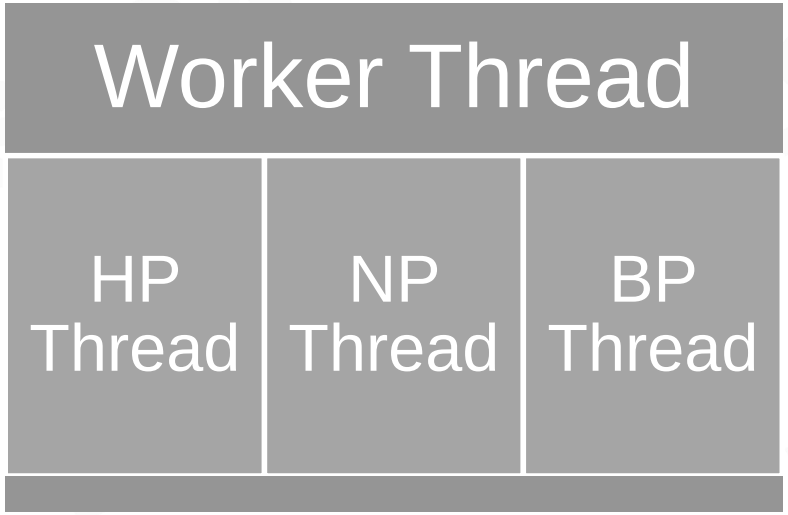
Unreal Parallel framework

Two types of threads

- Named Thread
  - Created by other systems and attached to parallel framework
- Worker Thread
  - Three priorities: high, middle and low
  - The number is determined by the number of CPU cores



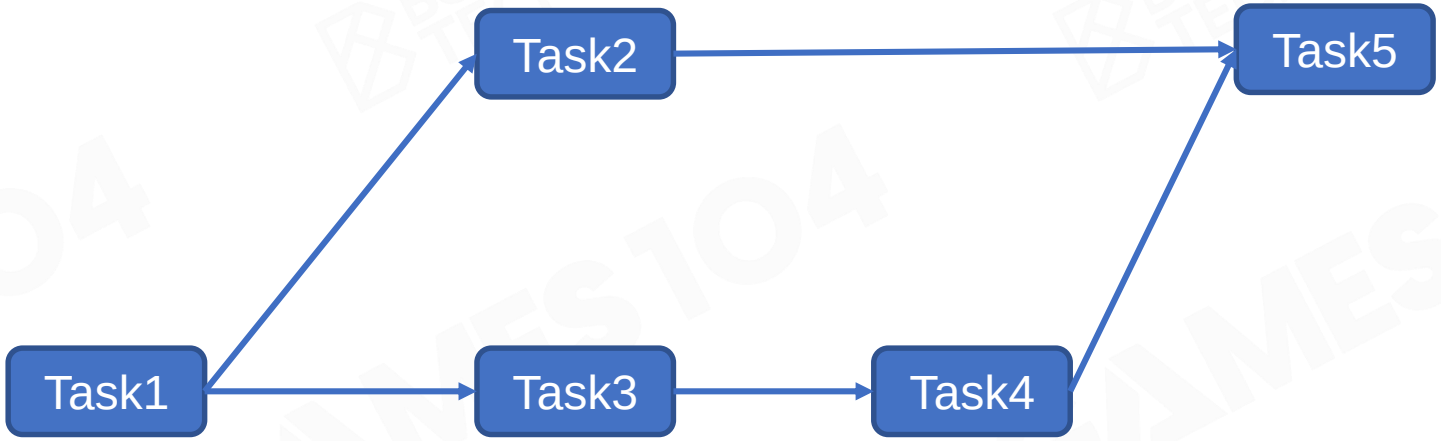




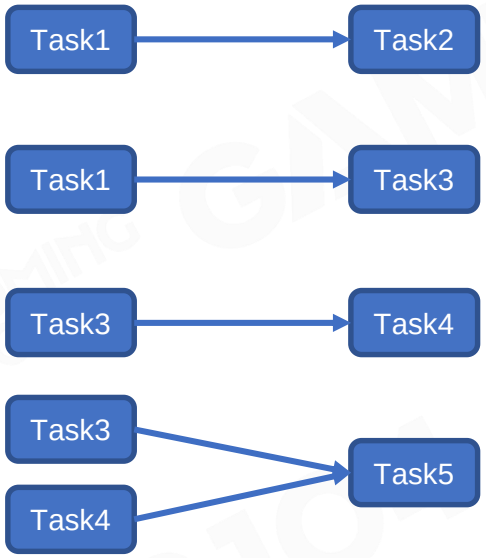
Task graph

A directed acyclic graph

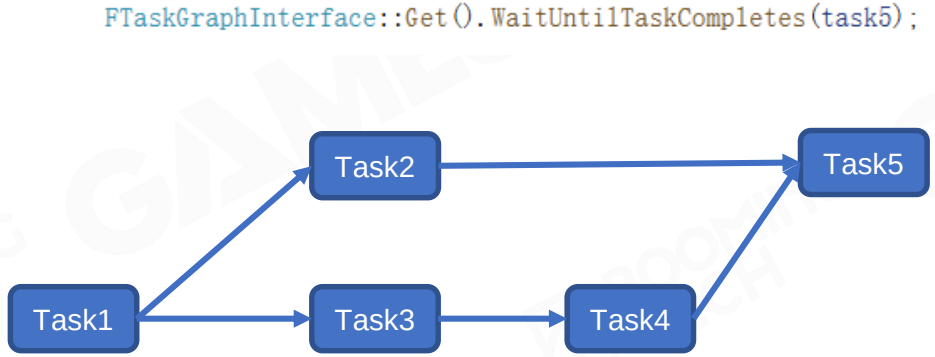
- Node→Task
- Edge→Dependency



Building Task Graph by Links




```
FGraphEventRef task1 = TGraphTask<FTaskGraphTestTask>::CreateTask().ConstructAndDispatchWhenReady();

FGraphEventArray task2_prerequistes;
task2_prerequistes.Add(task1);
FGraphEventRef task2 = TGraphTask<FTaskGraphTestTask>::CreateTask(&task2_prerequistes).ConstructAndDispatchWhenReady();

FGraphEventArray task3_prerequistes;
task3_prerequistes.Add(task1);
FGraphEventRef task3 = TGraphTask<FTaskGraphTestTask>::CreateTask(&task3_prerequistes).ConstructAndDispatchWhenReady();

FGraphEventArray task4_prerequistes;
task4_prerequistes.Add(task3);
FGraphEventRef task4 = TGraphTask<FTaskGraphTestTask>::CreateTask(&task4_prerequistes).ConstructAndDispatchWhenReady();

FGraphEventArray task5_prerequistes;
task5_prerequistes.Add(task2);
task5_prerequistes.Add(task4);
FGraphEventRef task5 = TGraphTask<FTaskGraphTestTask>::CreateTask(&task5_prerequistes).ConstructAndDispatchWhenReady();

FTaskGraphInterface::Get().WaitUntilTaskCompletes(task5);
```

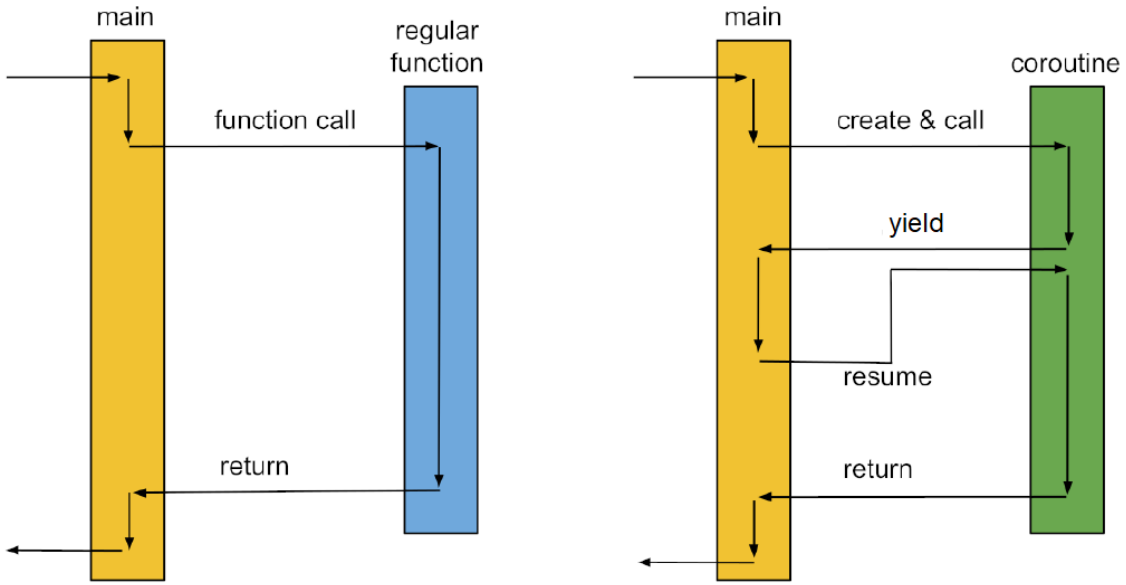


Job System

Coroutine

Allows for multitasking by creating ***jobs*** that run inside of ***coroutines***

- ***Coroutine*** is a lightweight execution context (include a user provided stack, registers...)
  - Execution is collaborative, means a coroutine can switch to another interactively



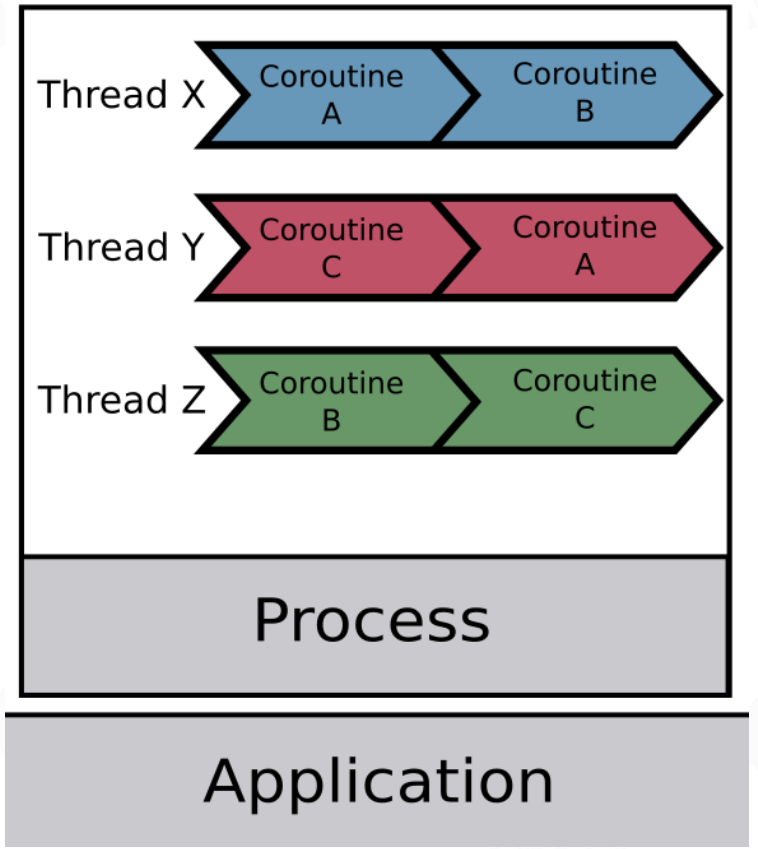
Coroutine vs. Thread

Coroutine

- Scheduled by programmers
- To be executed within a thread
- Context switch is faster without kernel switch

Thread

- Scheduled by operating system
- Resides in a process
- Context switch is costly with kernel switch



Stackful Coroutine

Coroutine owns an independent runtime stack which is reserved after yield

- Enable to yield from within a nested stackframe
- Use local variables just like normal functions

```
void main()
{
    doCoroutine(stackfulCO);
    ...
}
```

```
void stackfulCO()
{
    subroutine();
}
```

```
void subroutine()
{
    int a = 3;
    // OK, stack is reserved
    yield();
    // stack is restored

    assert(a == 3); // never fire
}
```

Stackless Coroutine

Coroutine has no independent runtime stack to be reserved when yield

- Only the top-level routine may yield (subroutines have no idea where to return without stack)
- The data that is required to resume execution should be stored separately from the stack

```
void main()
{
    doCoroutine(stacklessCO);
    ...
}
```

```
void stacklessCO()
{
    subroutine();
    int a = 3;
    // OK
    yield();
    // resume with a different stack

    // CAN'T, 'a' is not in the stack
    assert(a == 3);
}
```

```
void subroutine()
{
    // CAN'T
    yield();
}
```

Stackful vs. Stackless Coroutine

Stackful Coroutine

- More powerful with enable to yield from within a nested stackframe
- Needs more memory to reserve stacks for each coroutine
- Coroutine context switch takes more time

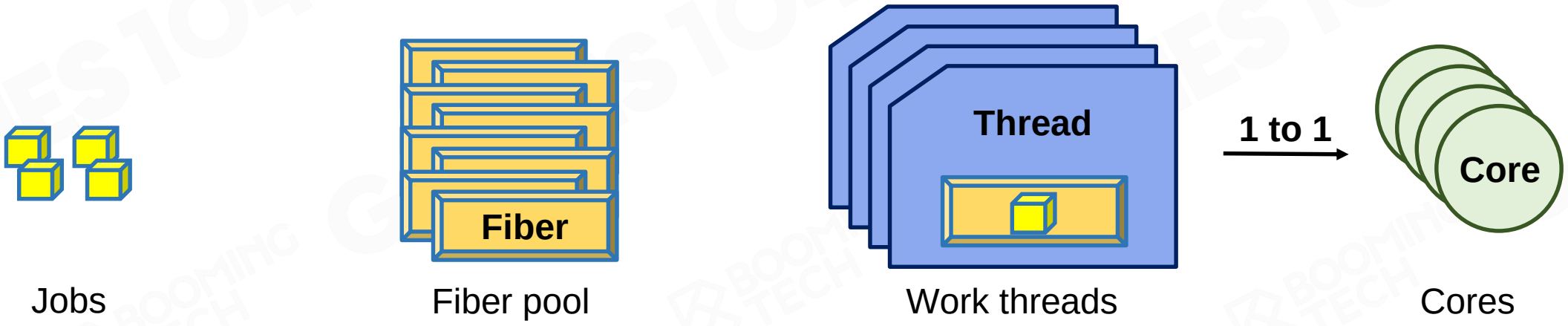
Stackless Coroutine

- Unable to yield from within a subroutine
- More difficult to use without a stack to reserve data
- No extra memory needed for coroutine's stack
- Faster context switch

Fiber-based Job System

Allows for multitasking by creating ***jobs*** instead of threads

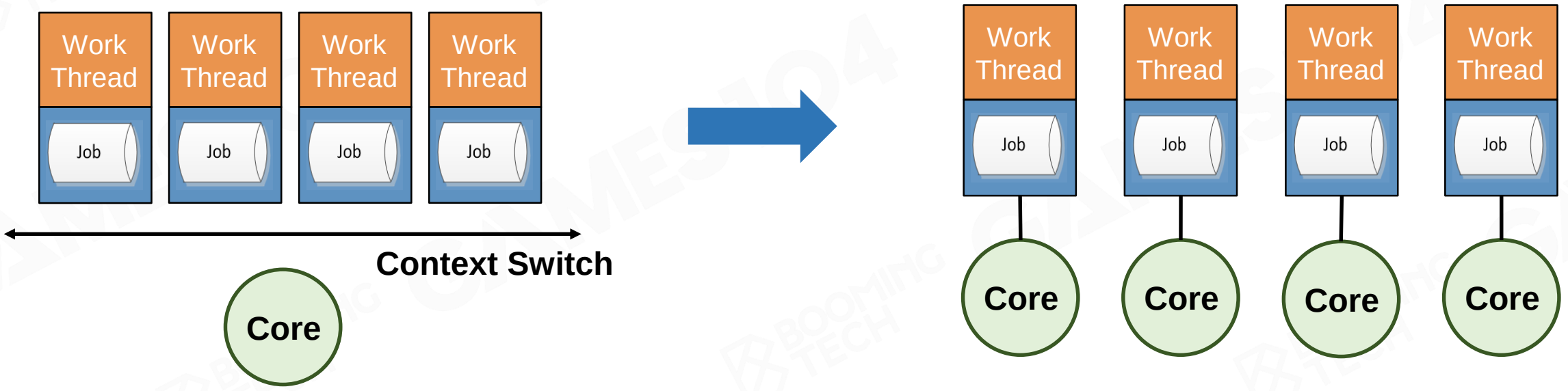
- ***Fiber*** is like ***coroutine*** except that fiber is scheduled by a scheduler
- Thread is the execution unit while fiber is the context
  - One thread for each processor core to minimize the context switch overhead
- Job is executed within the context of a fiber



One Work Thread for One Core

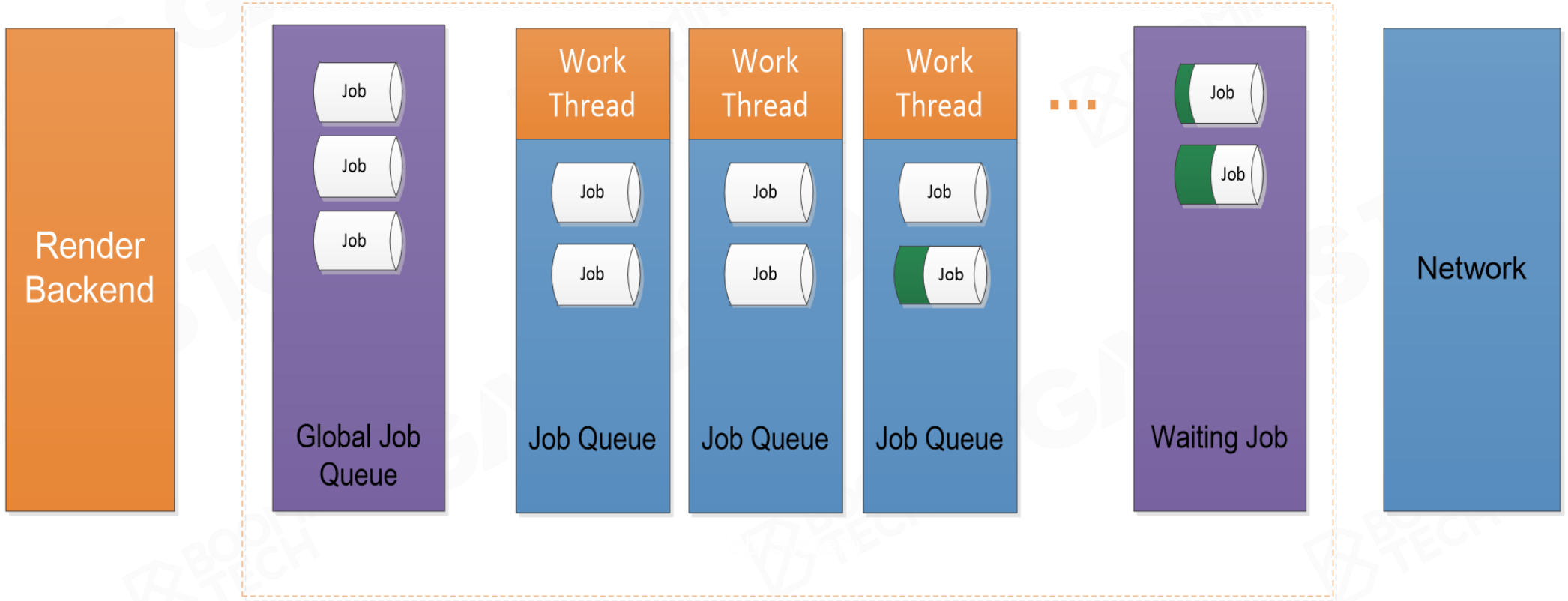
To minimize the overhead of thread context switch

- Multiple work threads for a single core still suffers from context switch
- One work thread for each core eliminates context switch

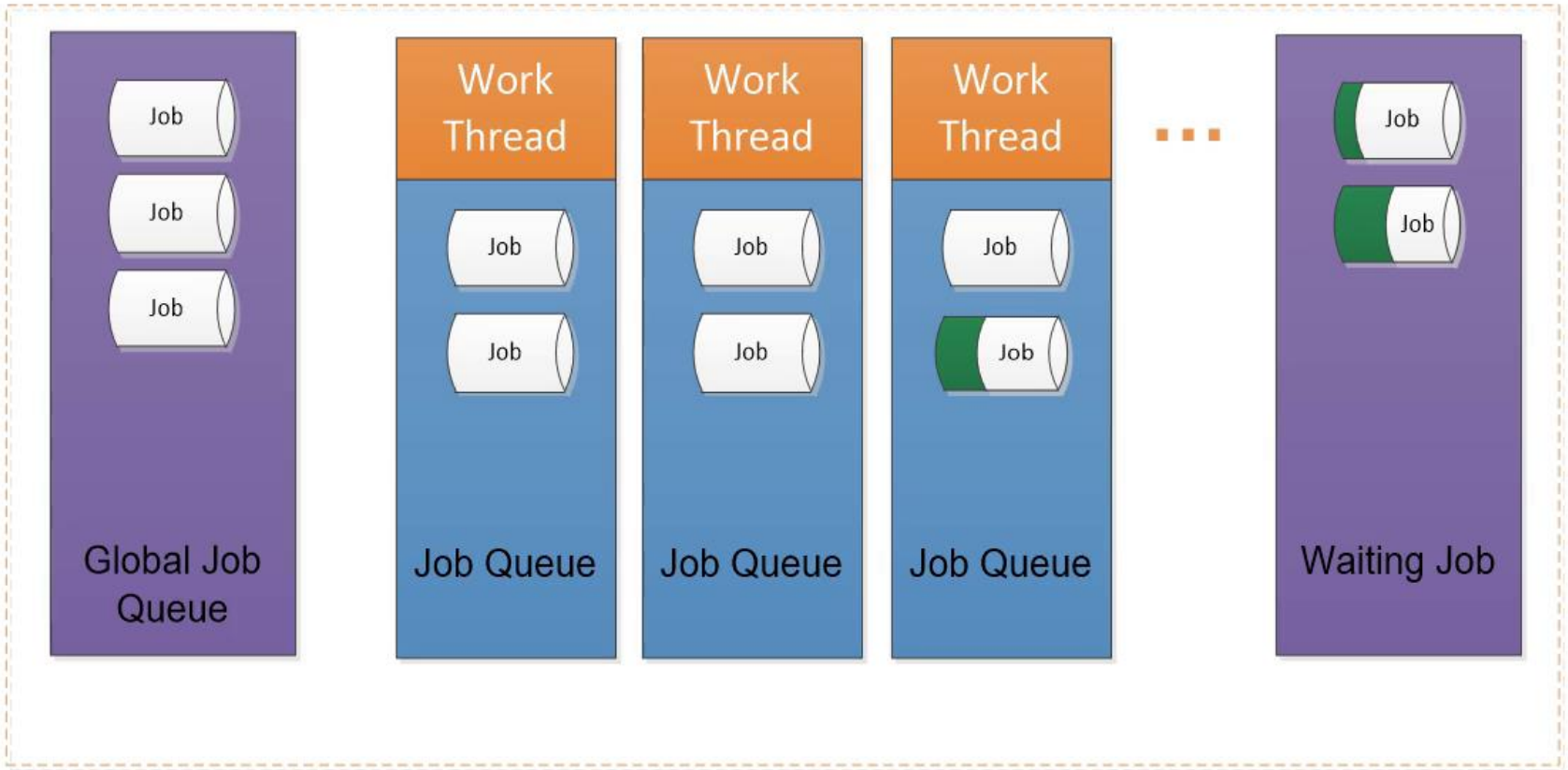


Fiber-based Job System

- Thread is the execution unit while **fiber** is the context
- Job is executed within a **fiber**



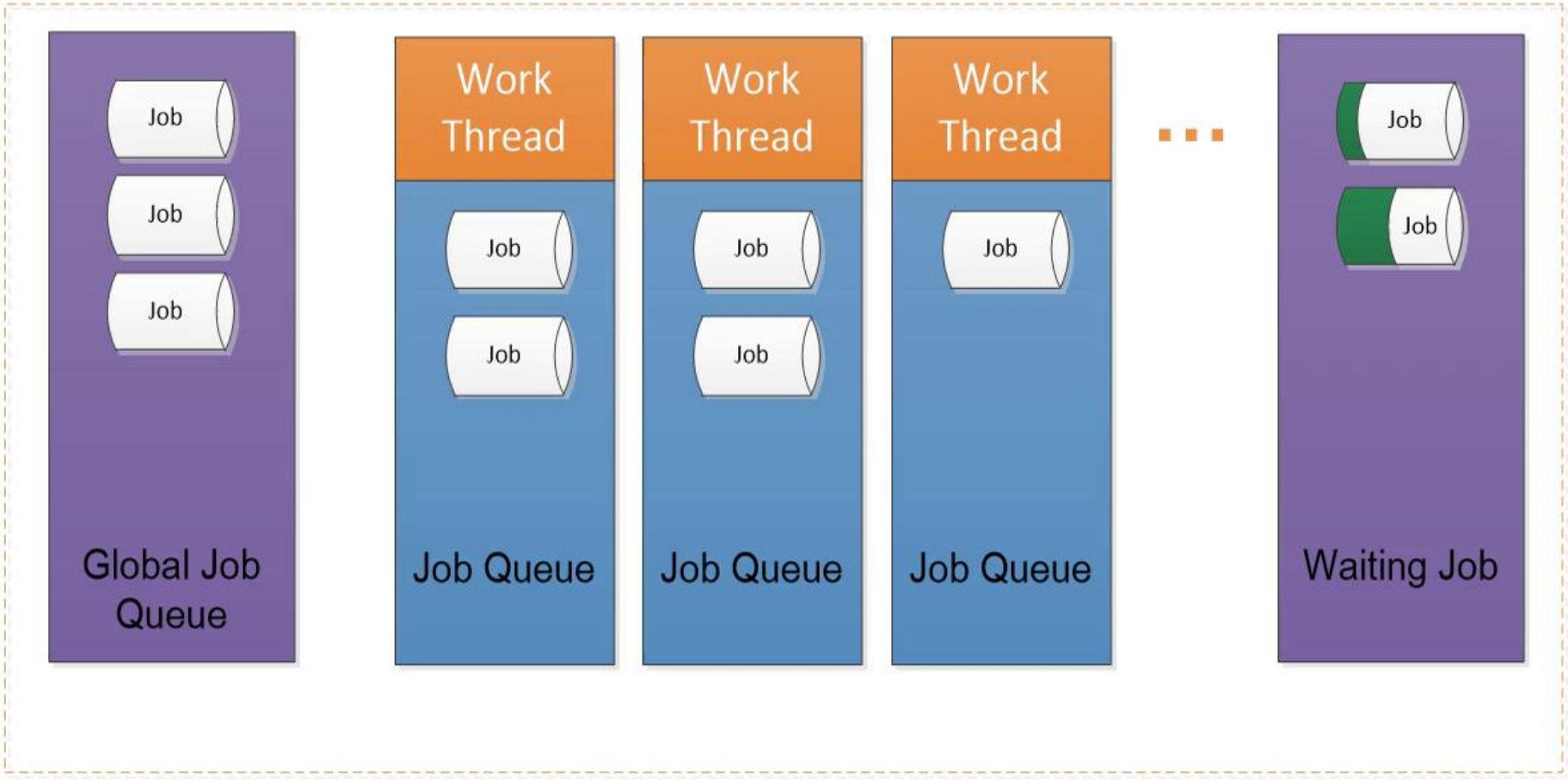
Job Scheduler - Global Job



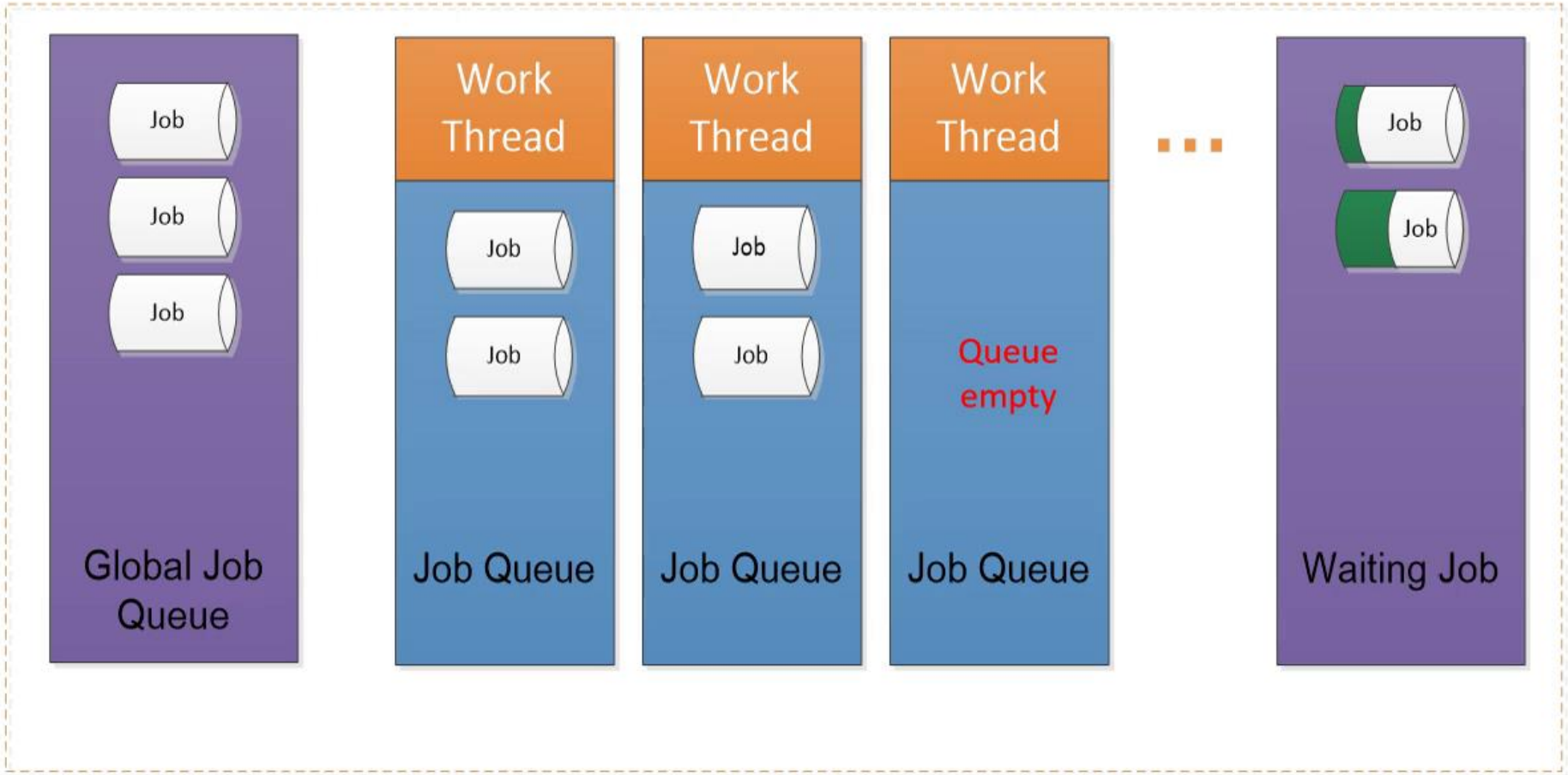
LIFO and FIFO Mode

- Schedule Model
  - First In First Out (FIFO)
  - Last In First Out (LIFO)
- LIFO Mode
  - In most case, job dependency is tree like
  - Some system add jobs occasionally but wait them immediately

Job Scheduler - Job Dependency



Job Scheduler - Job Stealing



Pros and Cons of Job System

Pros

- Easy to implement task schedule
- Easy to handle task dependency
- Job stack is isolated
- Avoid frequency context switch

Cons

- C++ does not natively support fiber
- Implementation is different between OS
- Has some restrictions(thread_local invalid)

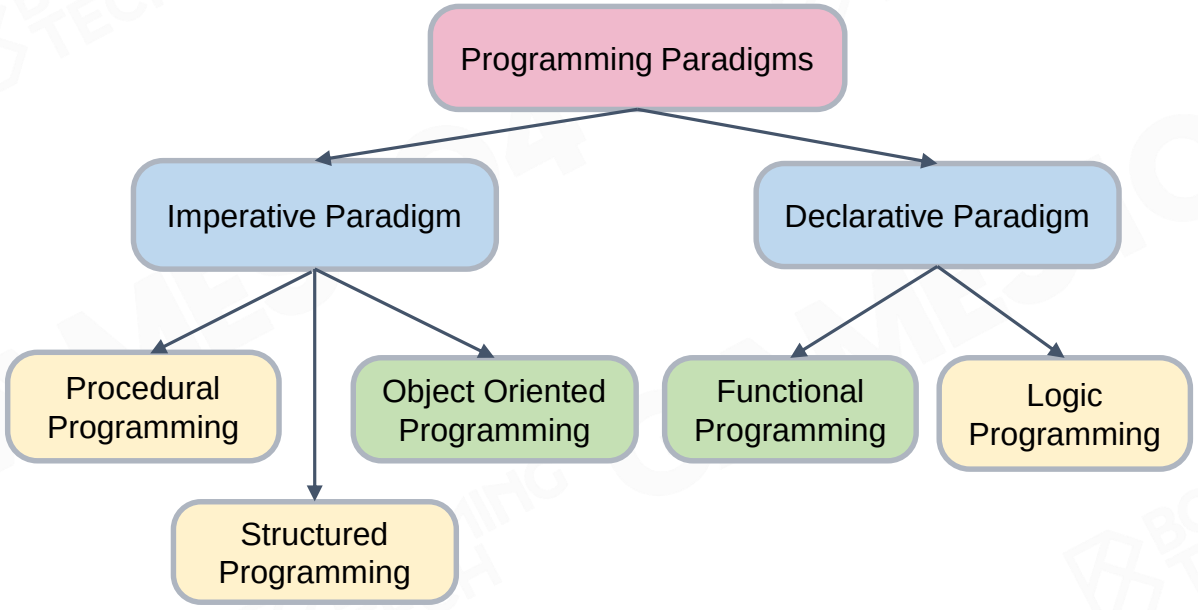
Programming Paradigms

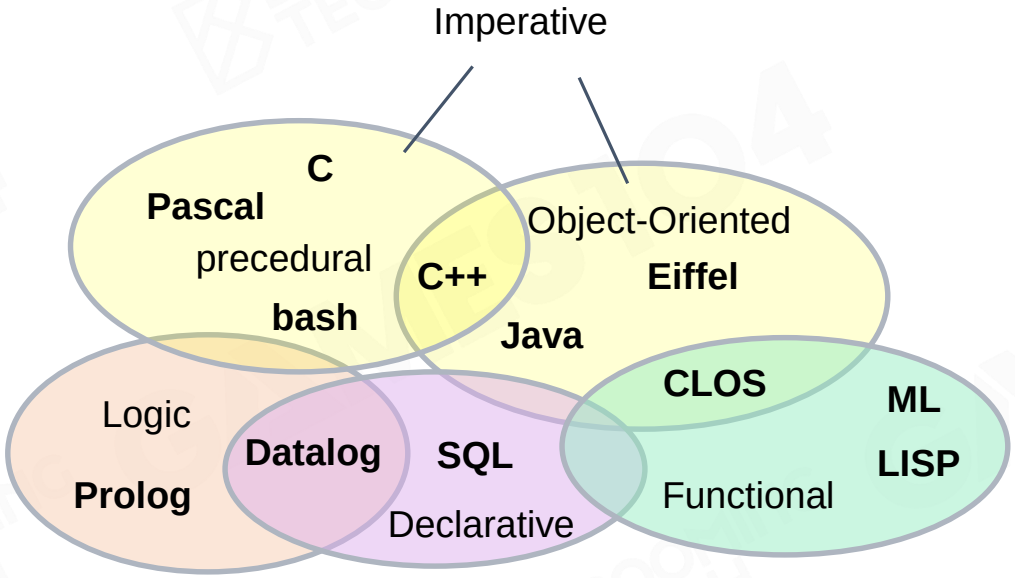
Procedure-oriented Programming

Object-oriented Programming

Programming Paradigm of Game Engine

- There are many different programming paradigms
- In practice, some paradigms are widely used
- Programming languages aren't always tied to a specific paradigm





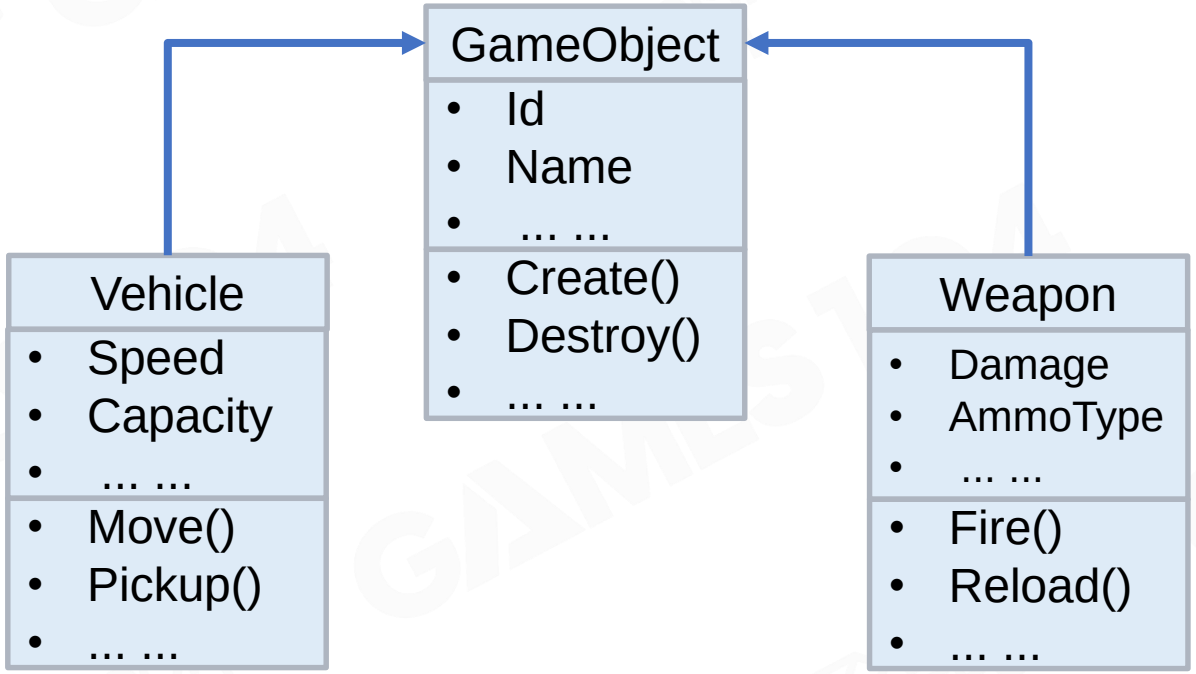
Procedural Oriented Programming (POP)

- Follows a step-by-step approach to break down a task into a collection of variables and routines (or subroutines) through a sequence of instructions

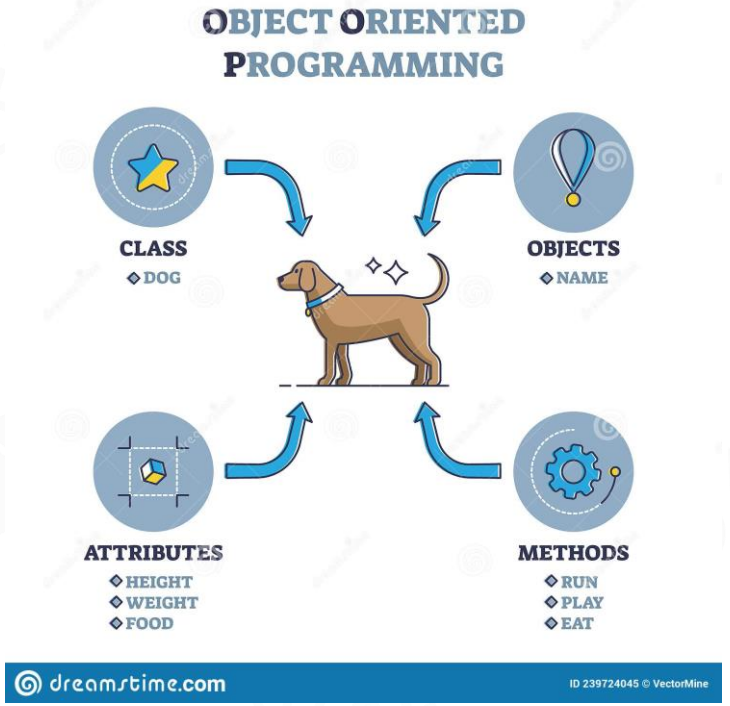
- **Impossible to write a game engine in this way**
  - Data is not well maintained.
  - A co-relation with real-world objects is difficult

Object-Oriented Programming (OOP)

- Based on the concept of "**objects**", which can contain **data** and **code**
- It's natural for human to abstract from real world in an object-oriented way



An example of game object



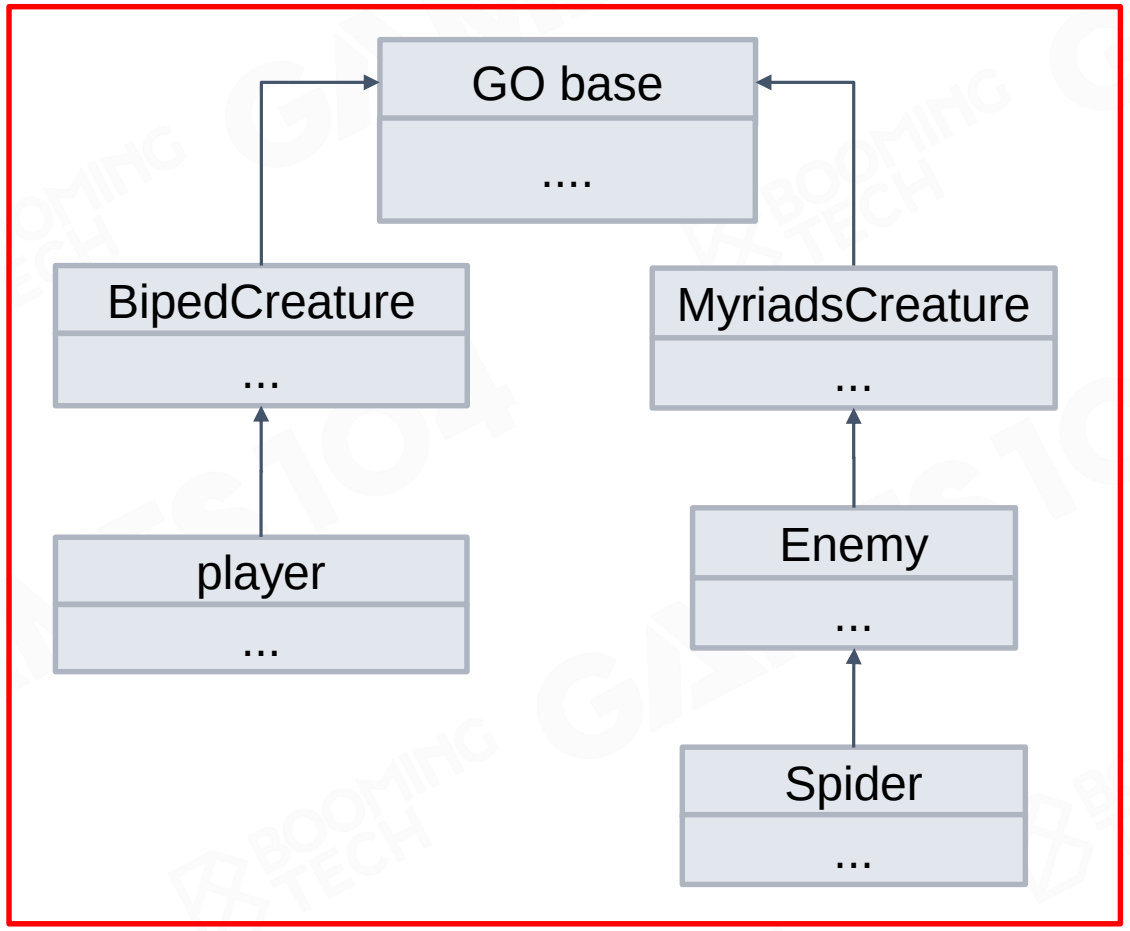
Problems of OOP : Where to Put Codes?

"Attacker.doDamageTo()",
or "Victim.receiveDamage()"?

"Player.attachTo()",
or "Enemy.isAttached()"?

Problems of OOP： Method Scattering in Inheritance Tree

- Hard to know which parent class has the method implementation



What happend when player attacks a spider enemy?

Need check many different classes and methods to find answer

Problems of OOP : Messy Based Class

```
class ENGINE_API AActor : public UObject
{
    ...
    const FTransform& GetTransform() const;
    const FTransform& ActorToWorld() const;
    FVector GetActorForwardVector() const;
    FVector GetActorUpVector() const;
    FVector GetActorRightVector() const;
    virtual void GetActorBounds(...) const;
    virtual FVector GetVelocity() const;
    float GetDistanceTo(const AActor* OtherActor) const;
    virtual void SetActorHiddenInGame(bool bNewHidden);
    bool GetActorEnableCollision() const;
    bool HasAuthority() const;
    UActorComponent* AddComponent(...);
    void AttachToActor(...);
    void DetachFromActor(const FDetachmentTransformRules& DetachmentRules);
    bool GetTickableWhenPaused();
    bool IsActorInitialized() const;
    void ReceiveAnyDamage(...);
    void GetOverlappingActors(...) const;
    virtual void SetLifeSpan(float InLifespan);
    virtual void Serialize(FArchive& Ar) override;
    virtual void PostLoad() override;
    ...
}
```

Parts of methods of a "messy base class"

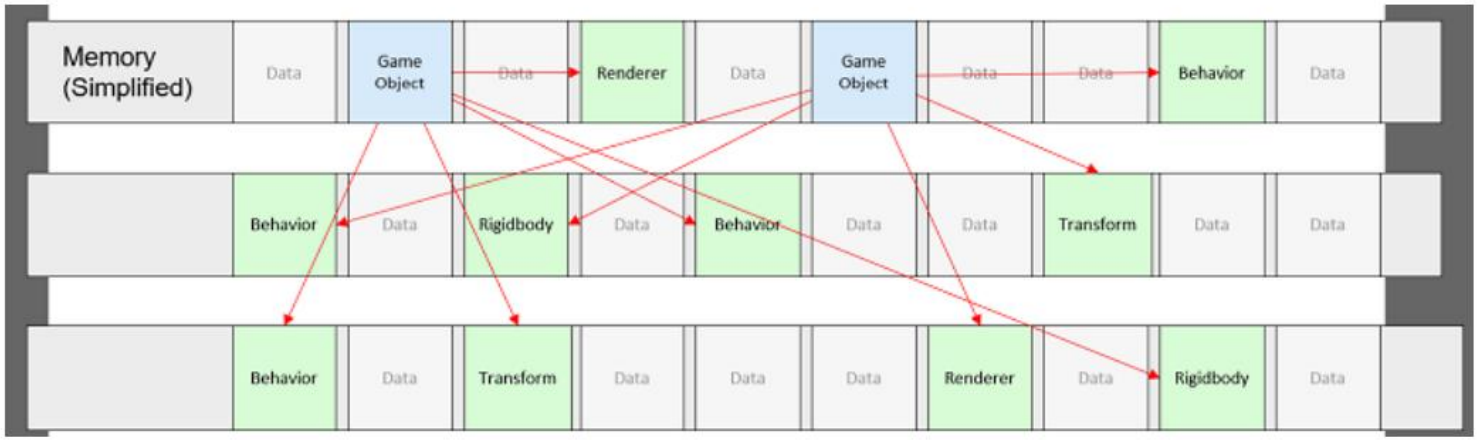
Find some methods in common?
Put it to the base class!

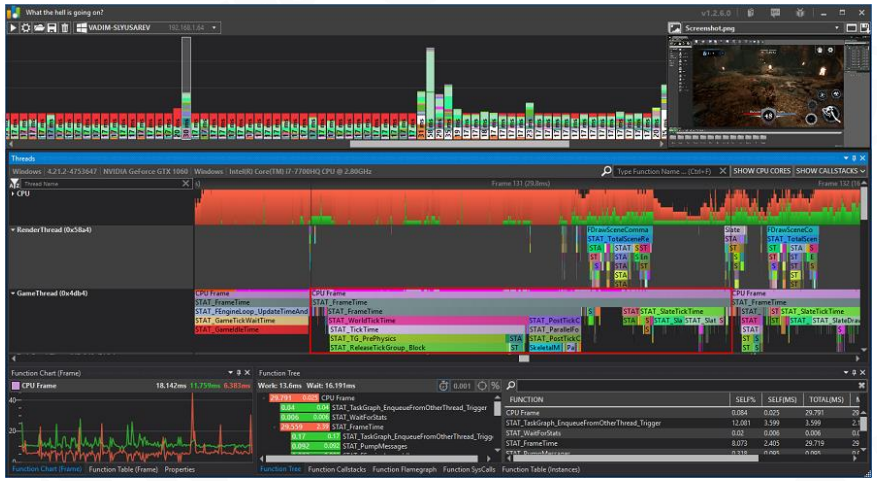
We get a messy base class

This is not the best OO design, and it certainly is possible to make a better one. But also, often code ends up being like this, even if no one wanted it that way.

Problems of OOP : Performance

- Memory scattering
- Jungle of virtual functions





Problems of OOP : Testability

- **Unit Testing**
  - OO designs often need a lot of setup to test

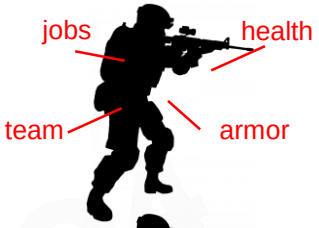
To test a **soldier.attack()** method

Setup several soldier objects

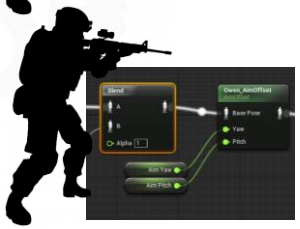
Which needs

- Initalize game object models

- Load basic attributes



- Setup animation tree



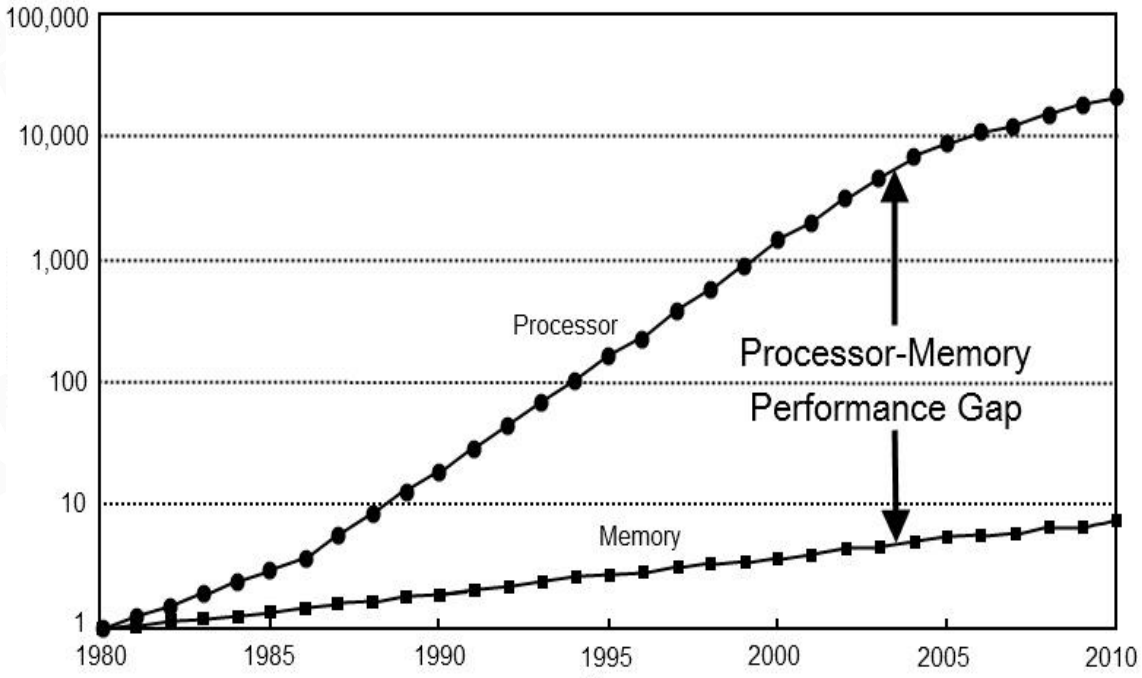
- Setup motor status

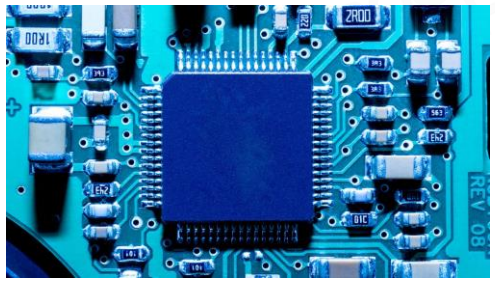
... ...

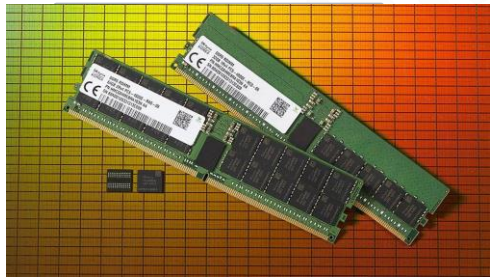
Data-Oriented Programming (DOP)

Processor-Memory Performance Gap

- Performance of memory grows much slowly than processor
- The gap is even larger which make **memory** becomes the **main bottleneck** of performance



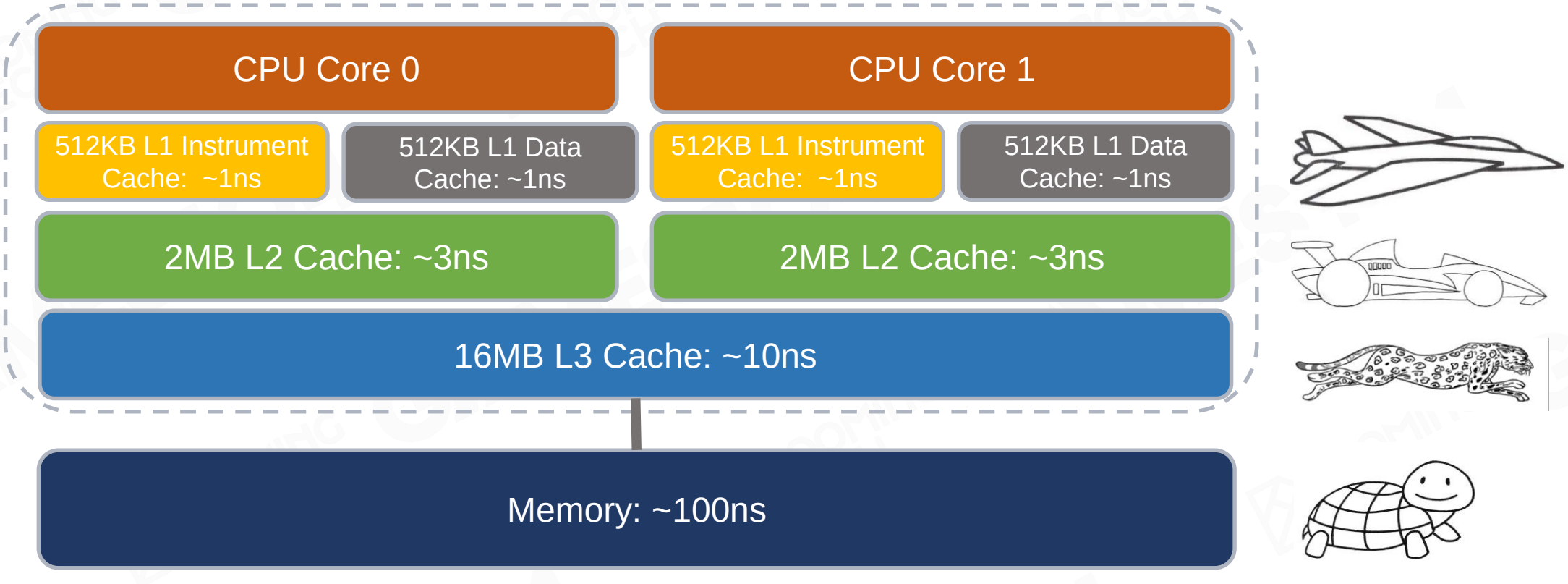




The Evolution of Memory - Cache

Add cache to speed up data reading

- L1: Ranges between 256KB to no more than 1MB, but even that is sufficient.
- L2: Usually a few megabytes and can go up to 10MB.
- L3: Larger than L1 and L2, varies from 16MB to 64MB, shared between all cores.



Principle of Locality

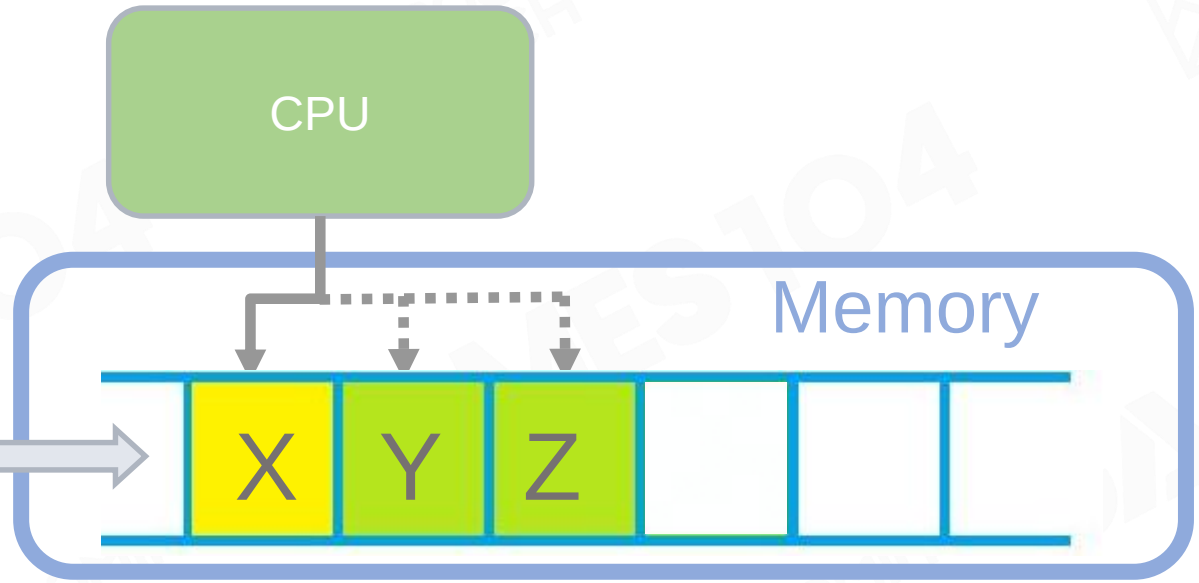
the tendency of a processor to access the same set of memory locations repetitively over a short period of time

Spatial Locality

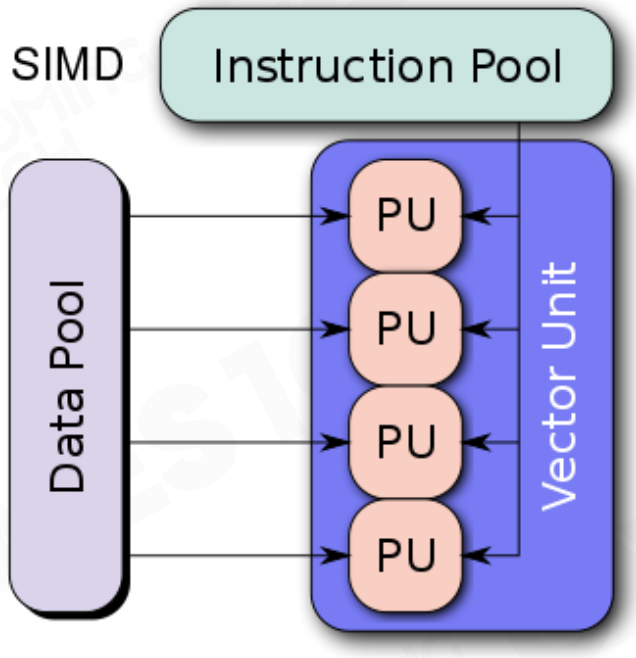
- The use of data elements within relatively close storage locations

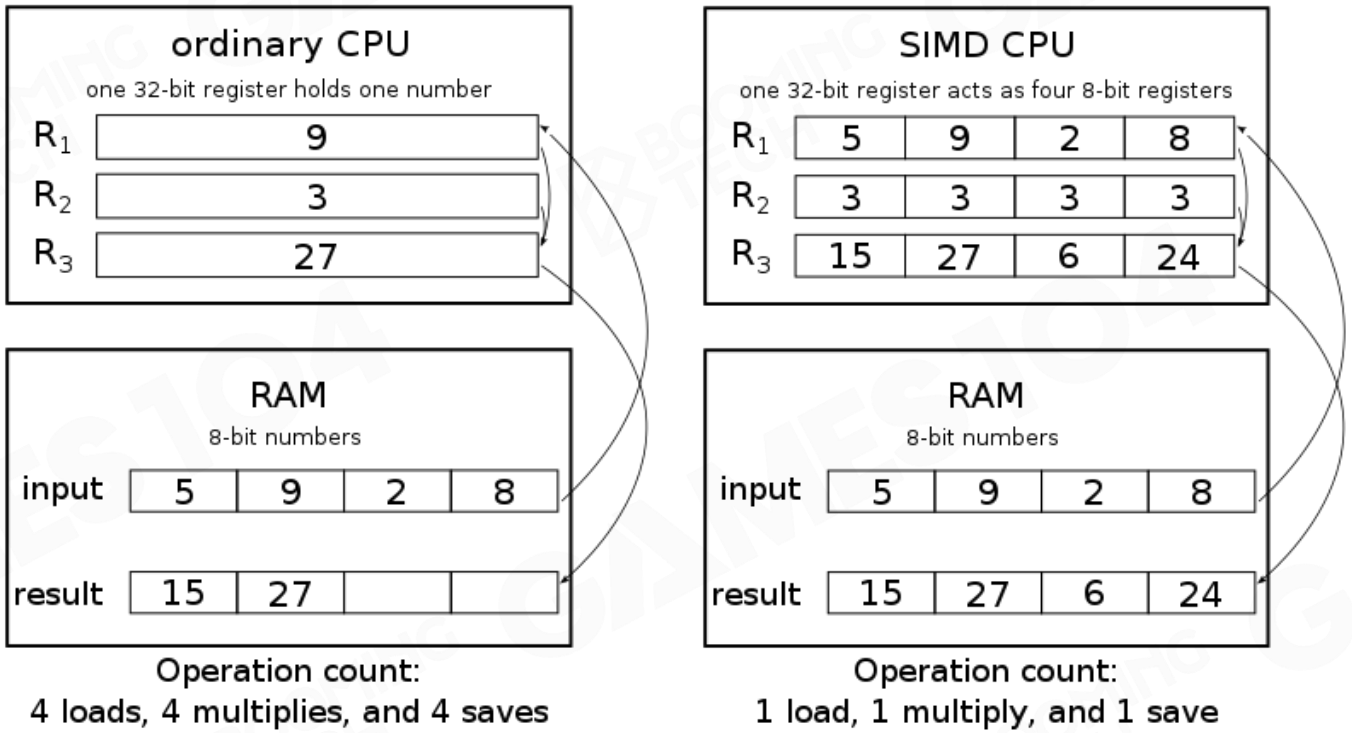
```
Vector3 v1, v2;
...
if(v1 == v2)
{
    ...
}
```

```
struct Vector3
{
    Float x, y, z;
}
```



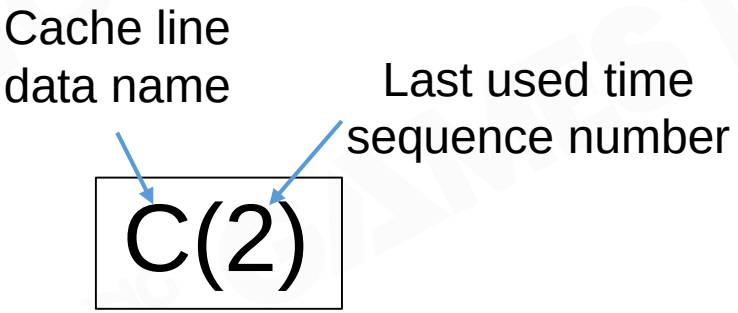
Single instruction multiple data (SIMD)

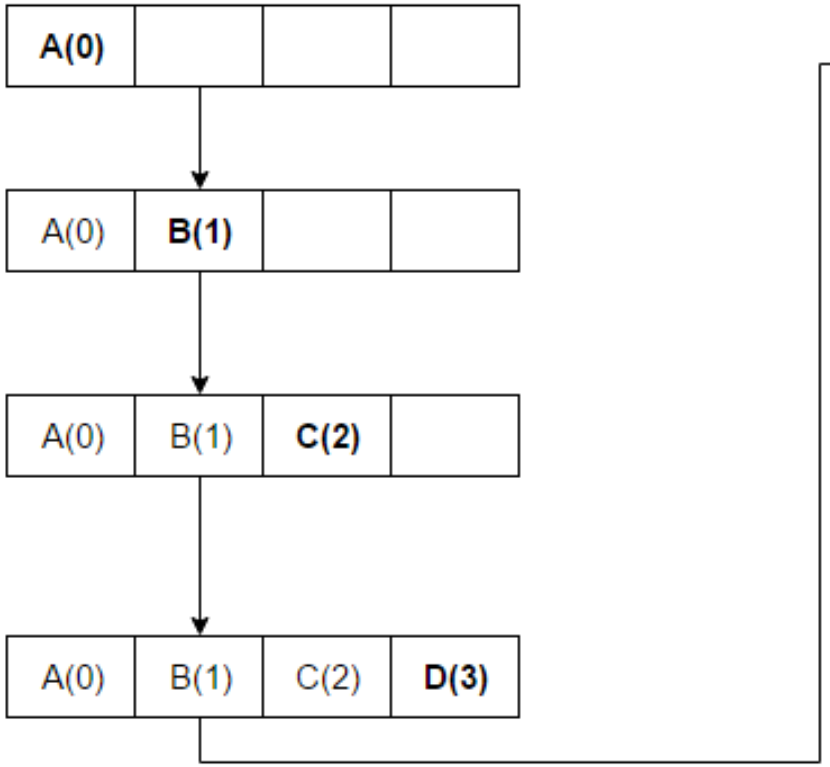


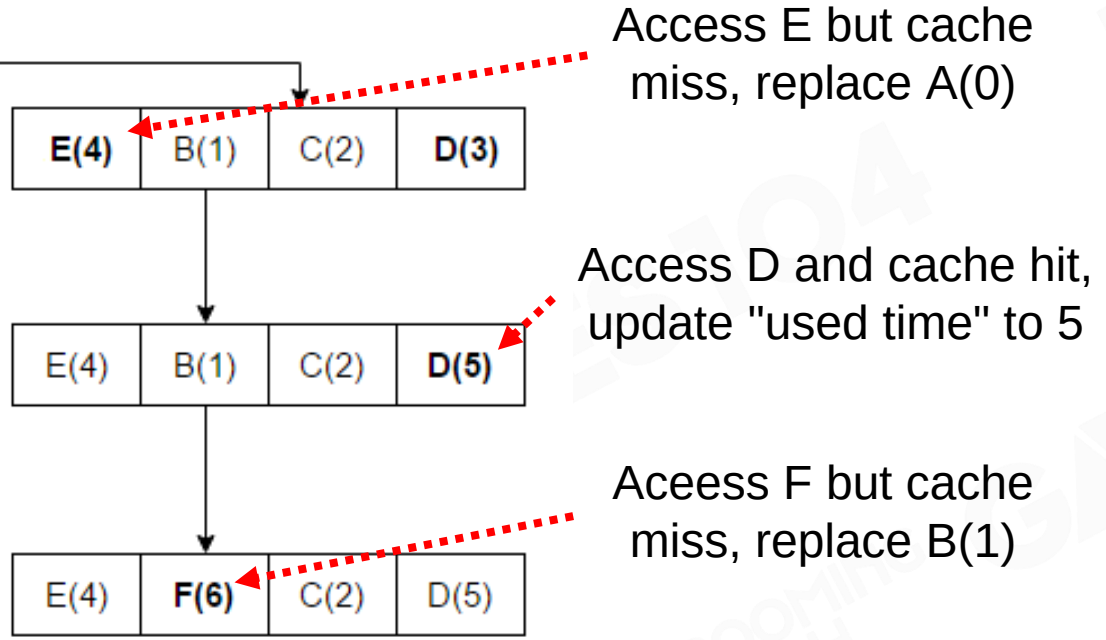


LRU (Least Recently Used)

- When cache is full, discards the **least recently used** cache-line first.
  - Record the "used time" of each cache line
  - Discard the most "oldest used" cache line each time
  - Update "used time" when access data of cache line



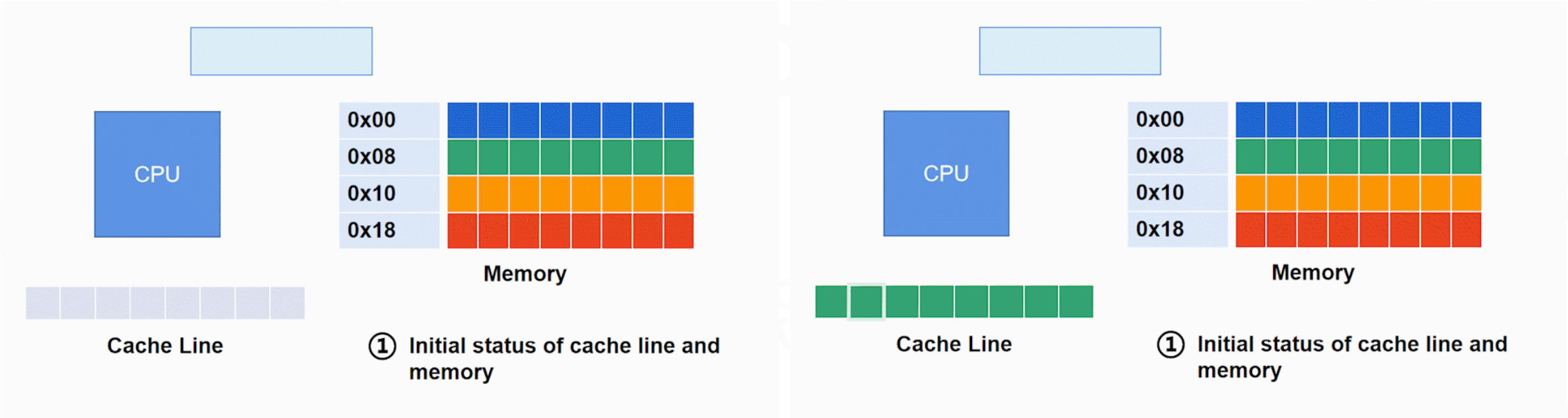




The data read sequence: **A, B, C, D, E, D, F**

Cache Line

- Data is transferred between memory and cache in **blocks of fixed size (typically 64 bytes)**, called cache lines or cache blocks.
- A cache can only **hold a limited number of lines**, determined by the cache size. For example, a 64 kilobyte cache with 64-byte lines has 1024 cache lines.
- Every time you load any memory at all, you are loading in a full cache line of bytes



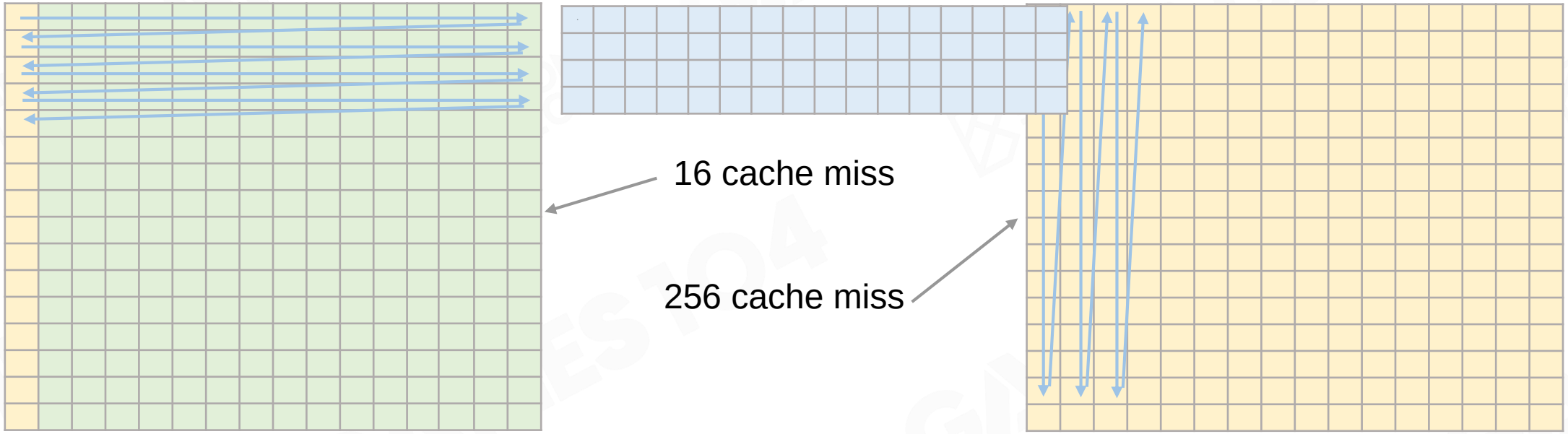
Cache Miss

Two way of iteration

Assume L1 cache with 4 cache lines
Each cache line is 64 bytes (16 integers)

Row-major order

Column-major order



- When cahce is full (loaded 4 rows), new rows will replace the oldest one
- When a elements not in cache, a whole row will be loaded

Data-Oriented Programming (DOP)

1. Data is all we have

Instructions are Data Too

```
int test()
{
    int a = 1;
    int b = 2;
    return a + b;
}
```

Code

1100011100000110
0000000000000000
0000000000000010

Data

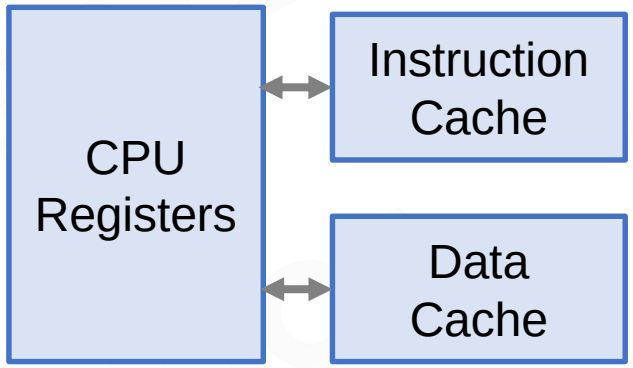
0101010101011010
1010001111100001
1000100011111001
1110000110101011
0000111001010101
01010101010.....

Elapsed Time : 31.539s

Clockticks:	85,288,127,932	
Instructions Retired:	131,378,197,067	
CPI Rate:	0.649	
MUX Reliability:	0.957	
Front-End Bound:	29.3%	of Pipeline Slots
Front-End Latency:	20.1%	of Pipeline Slots
ICache Misses:	7.1%	of Clockticks
ITLB Overhead:	3.1%	of Clockticks
Branch Resteers:	4.8%	of Clockticks
DSB Switches:	2.6%	of Clockticks
Length Changing Prefixes:	0.1%	of Clockticks
MS Switches:	3.1%	of Clockticks
Front-End Bandwidth:	9.3%	of Pipeline Slots
Bad Speculation:	8.2%	of Pipeline Slots
Back-End Bound:	21.5%	of Pipeline Slots
Retiring:	40.9%	of Pipeline Slots
Total Thread Count:	6	
Paused Time:	0s	

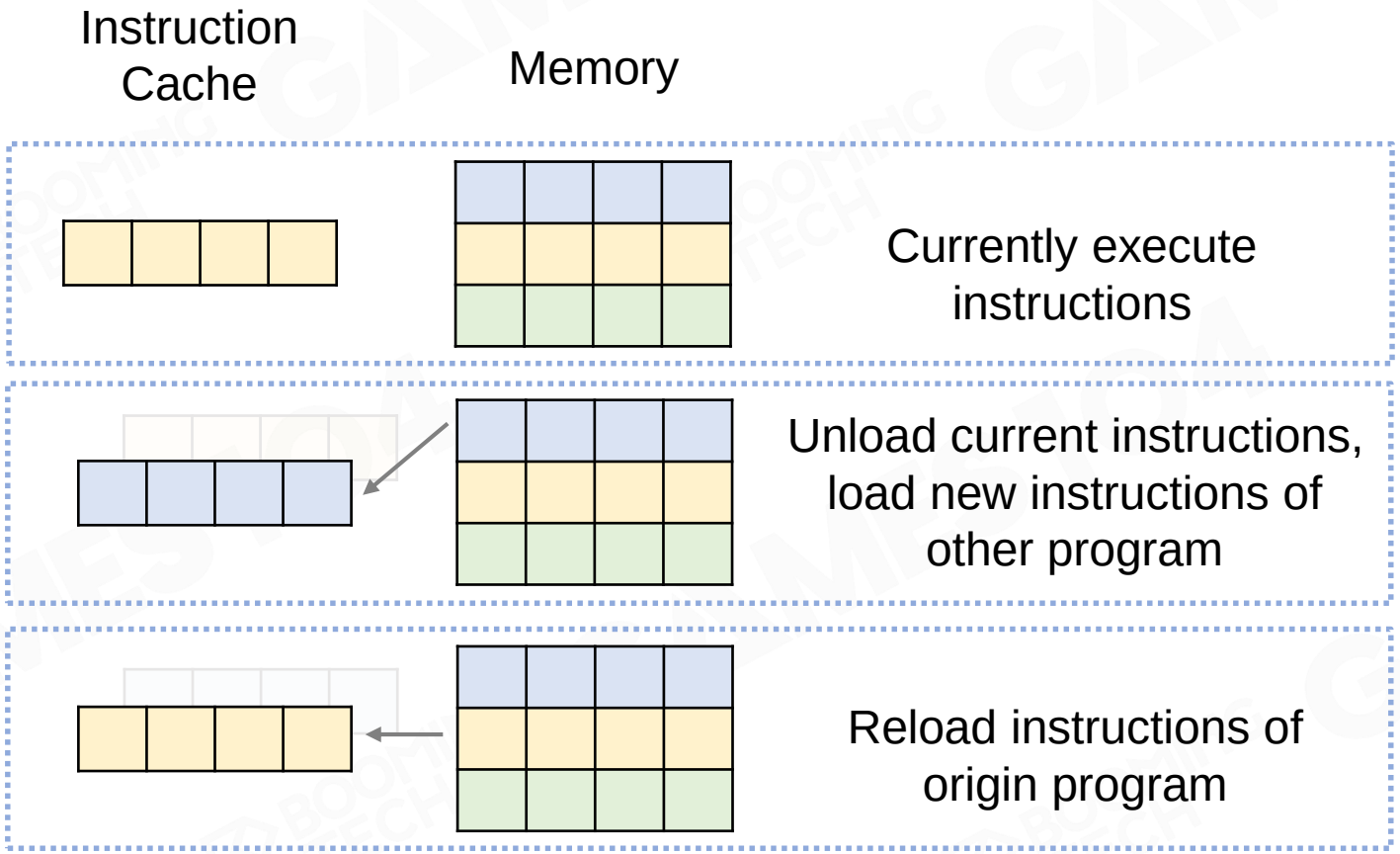
Keep Code and Data Tight in Memory

- Keep both code and data small and process in bursts when you can



CPU may switch to other program in runtime

Code may also have cache miss problem!



Performance-Sensitive Programming

Reducing Order Dependency

- The work being done because of a misprediction will have to be undone
- Never modify variables once they are initially assigned

These 2 parts of code will not be excuted in parallel

because variables a & b is used before

```
a = 2
b = a * 5
... ...
a = 4
b = a / 2
... ...
```

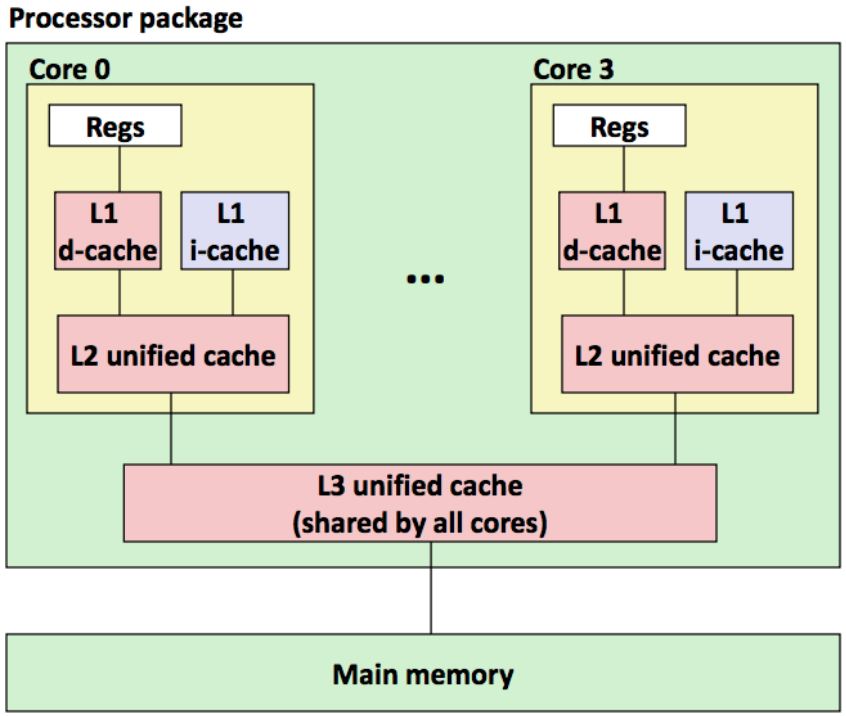
```
a = 2
b = a * 5
... ...
a2 = 4
b2 = a2 / 2
... ...
```

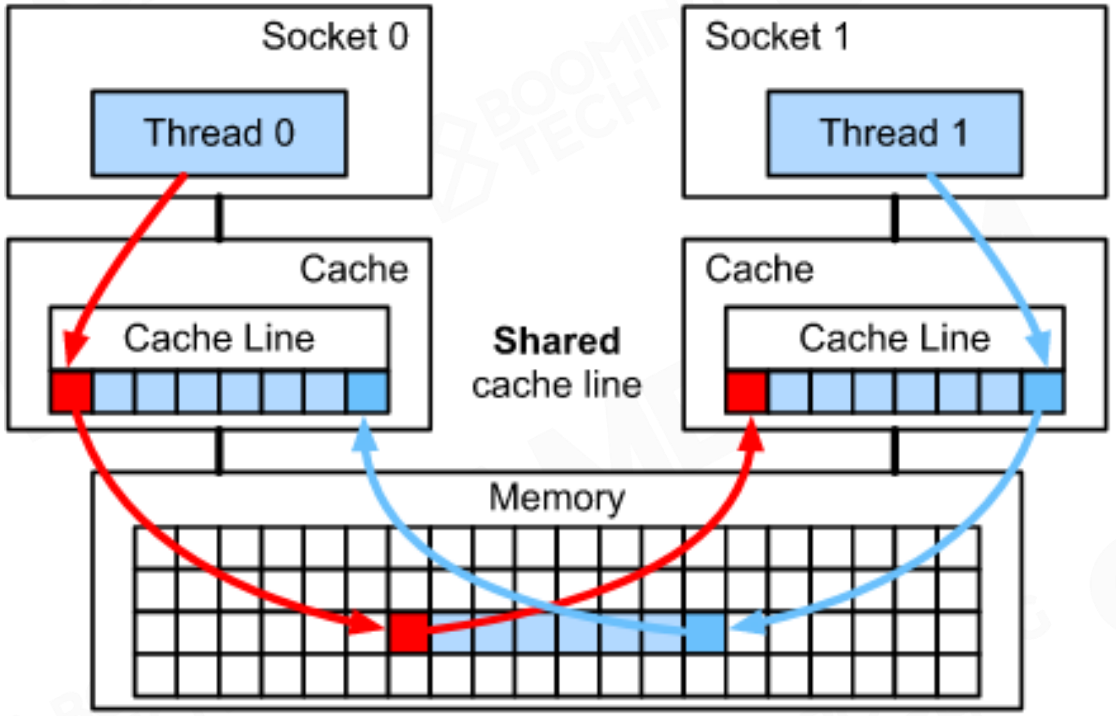
Compiler allow these 2 parts of code to execute in parallel

Actually, compiler use static single-assignment (SSA) to deal with simple situation like this

False Sharing in Cache Line

- Ensuring any rapidly updated variables are kept local to the thread
- Cache contension



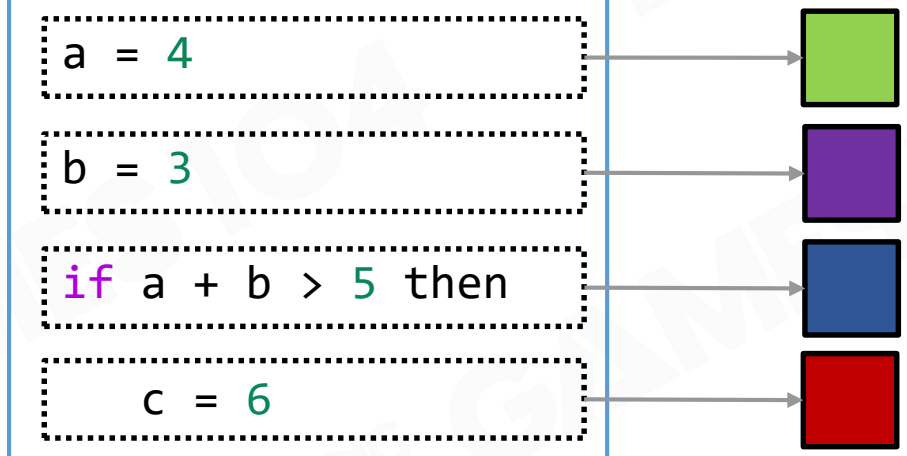


Branch prediction (1/3)

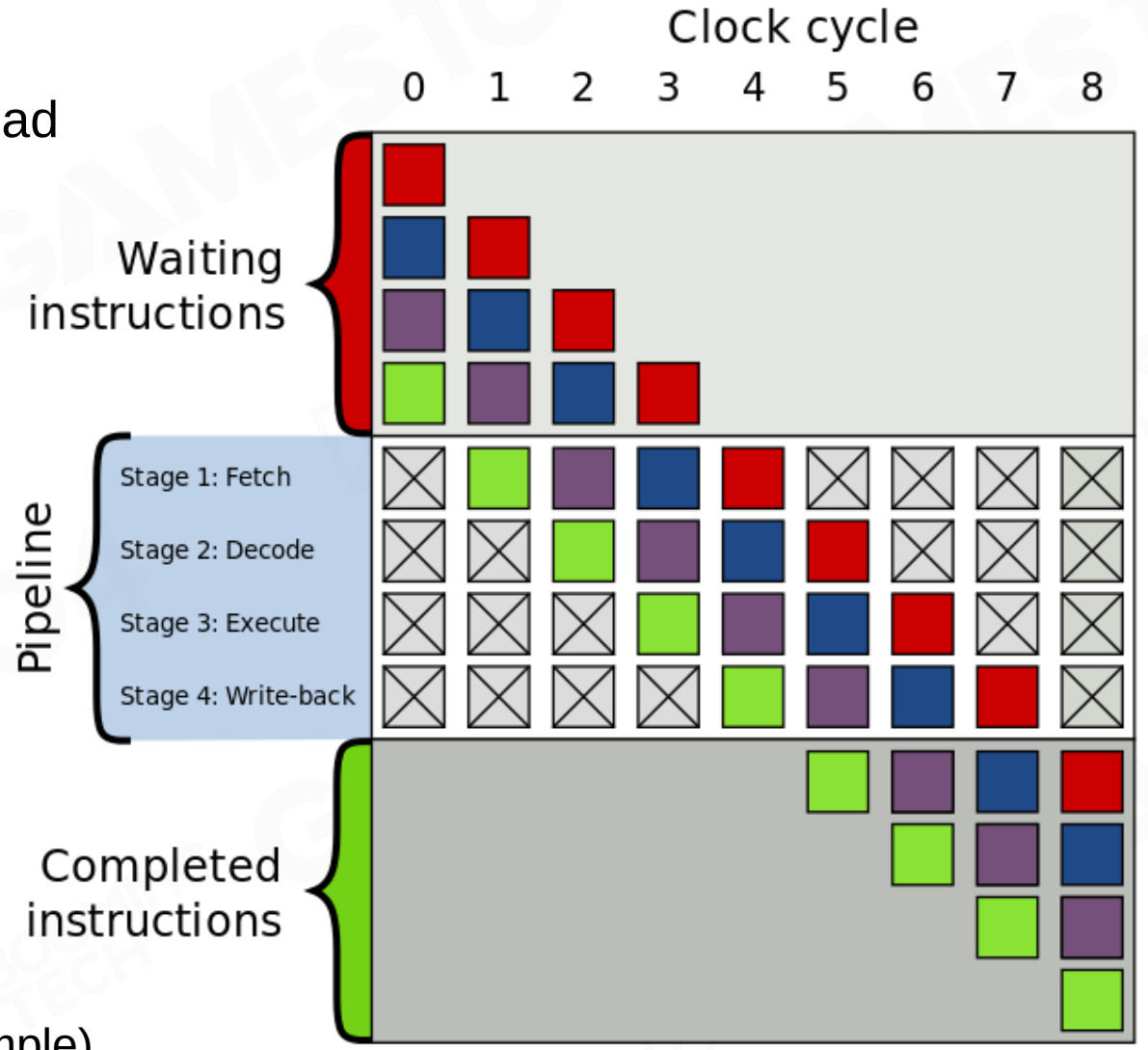
- CPU will prefetch instructions and data ahead
- Use branch prediction technics to decide what to prefetch

```
a = 4
b = 3
if a + b > 5 then
    c = 6
```

Code



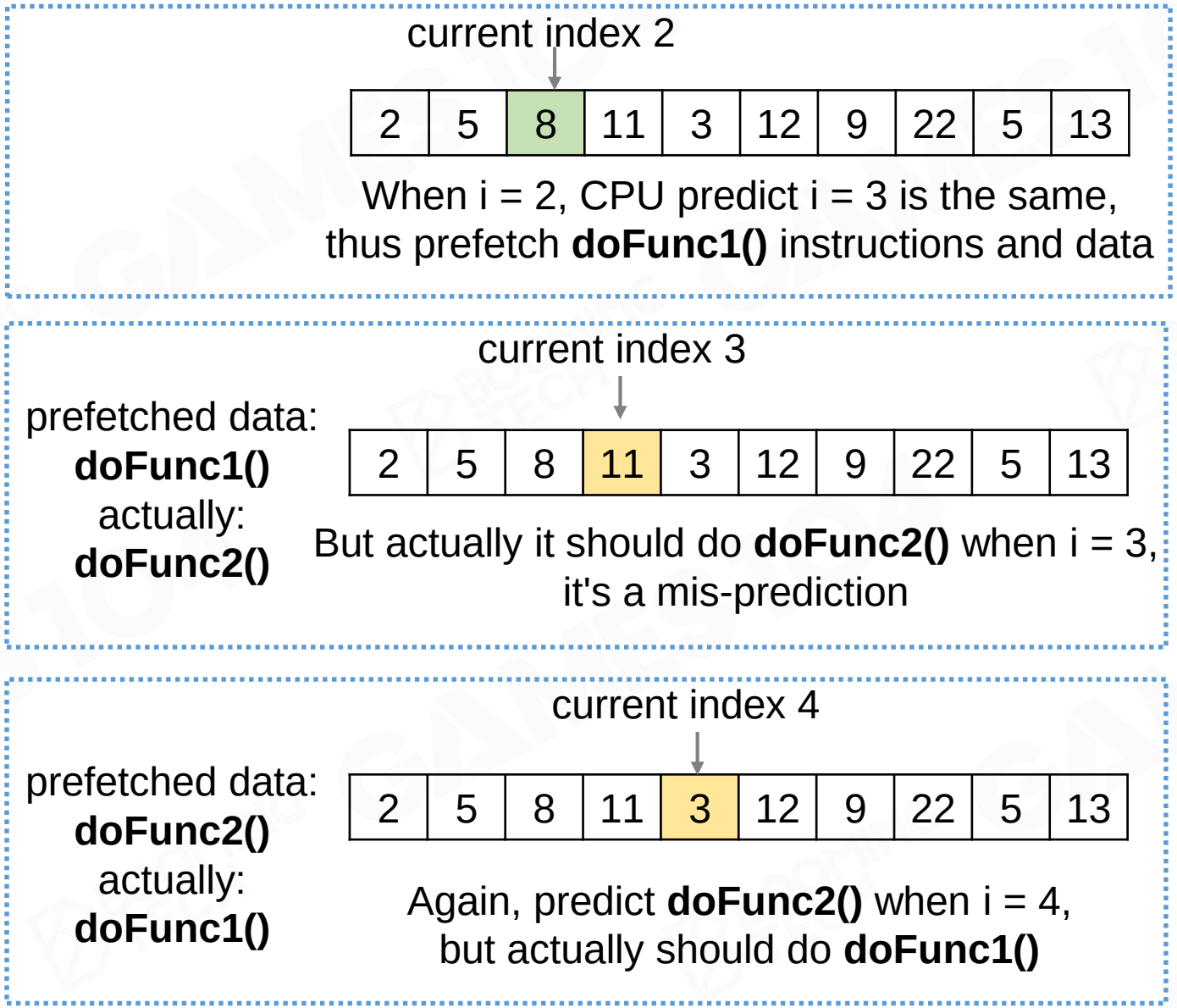
Instrument
(simplified for example)



Branch prediction (2/3)

- To avoid branch mis-prediction

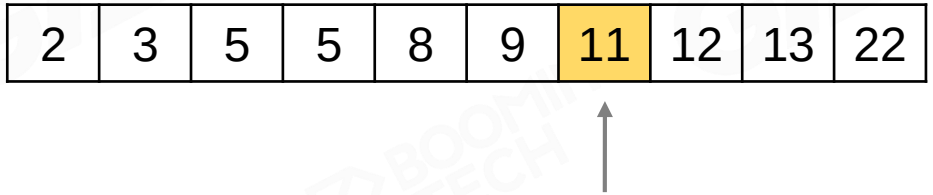
```
int a[10] = {2,5,8,11,3,12,9,22,5,13};
for (int i = 0; i < 10; i ++)
{
    if (a[i] > 10)
    {
        doFunc1();
    }
    else
    {
        doFunc2();
    }
}
```



Branch prediction (3/3)

- To avoid branch mis-prediction

```
int a[10] = {2,3,5,5,8,9,11,12,13,22};
for (int i = 0; i < 10; i ++)
{
    if (a[i] > 10)
    {
        doFunc1();
    }
    else
    {
        doFunc2();
    }
}
```



If it's a sorted array, only 1 mis-prediction will occur

Existential Processing

Only processing existing elements rather than deciding whether should be processed on the fly

```
for actor in actor_array do
    if actor is alive then
        aliveFunc(actor)
    else
        deadFunc(actor)
    end
end
```

This code also faces branch prediction problems

Unlike the example before, actor_array changes every tick

```
for actor in alive_actor_array do
    aliveFunc(actor)
end

for actor in dead_actor_array do
    deadFunc(actor)
end
```

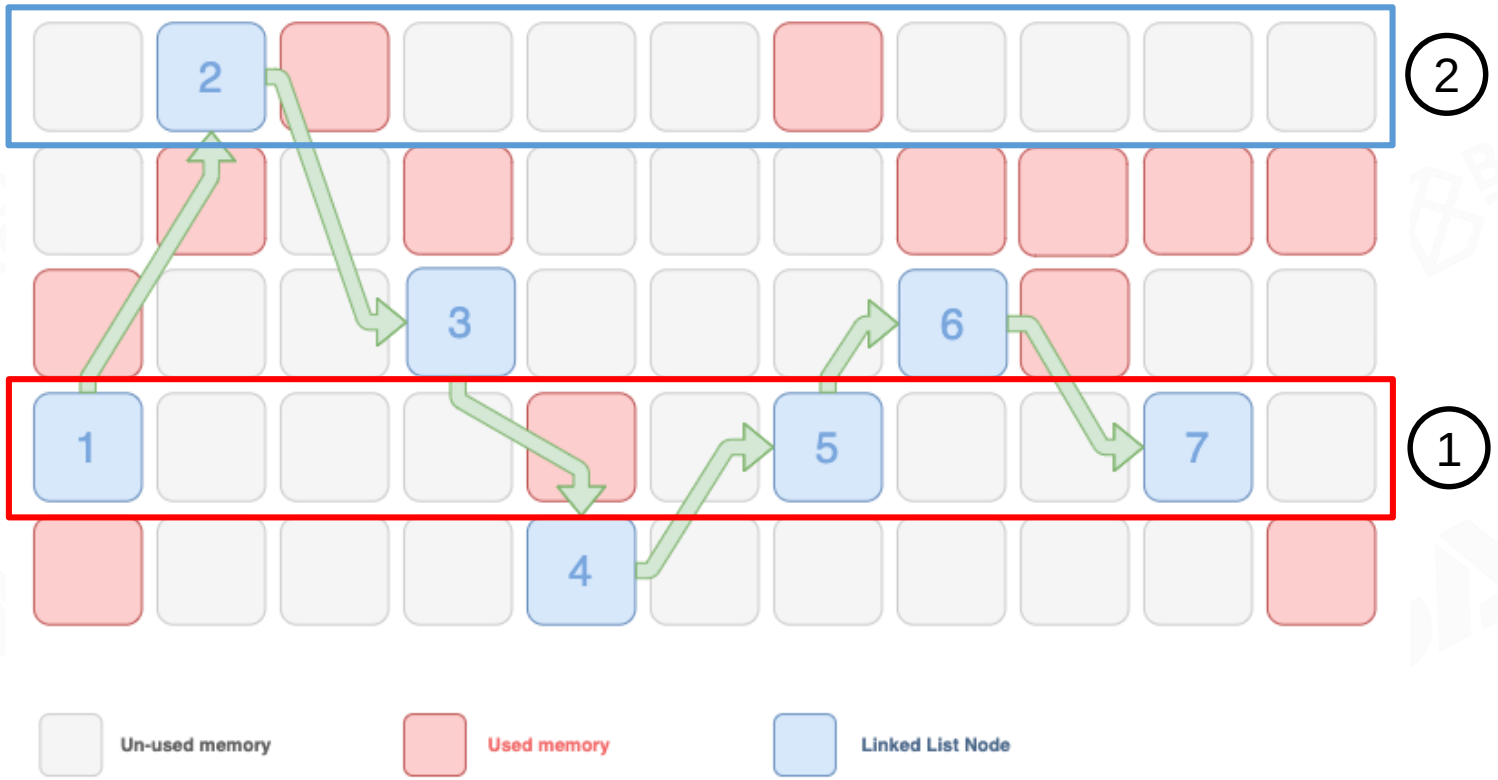
Completely avoid "if-else"

By maintaining 2 lists of different actors, we could avoid branch mis-precondition

Performance-Sensitive Data Arrangements

Reducing Memory Dependency

- (chained memory lookups/accesses by pointers)

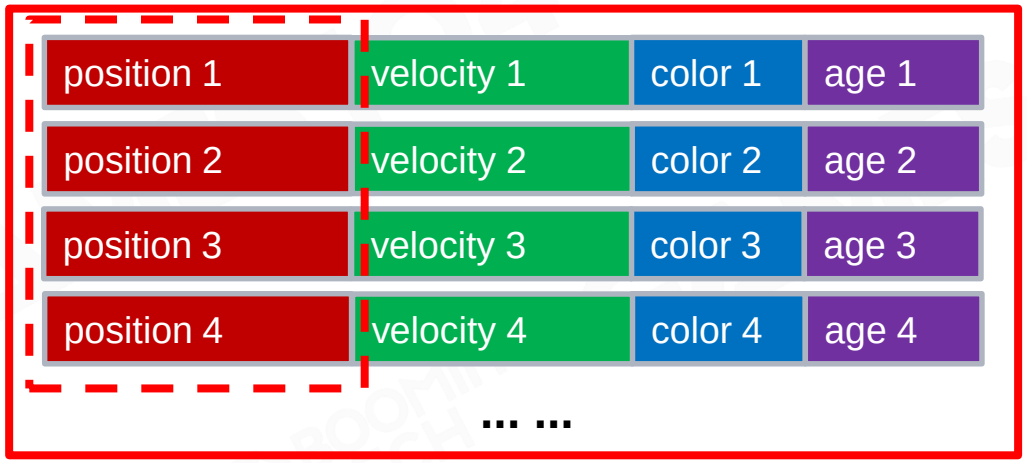


- Load the first cache line 1
- Get the next node address
- Cache miss
- Unload the old one, and load another cahce line 2
- Repeating

Array of Structure vs. Structure of Array

AOS

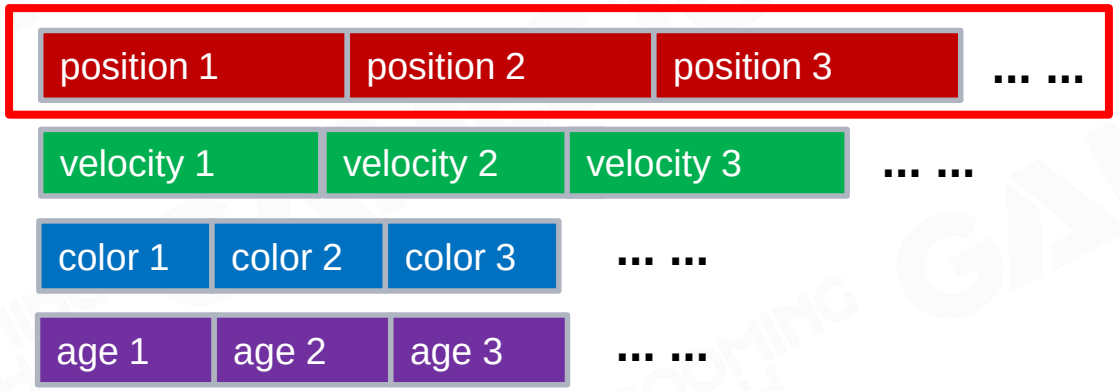
```
struct Particle{
    Vector3 position;
    Vector3 velocity;
    Color color;
    float age;
    //...
} Particle[N];
```



If we want to read the **position** of all particles, SOA has better performance

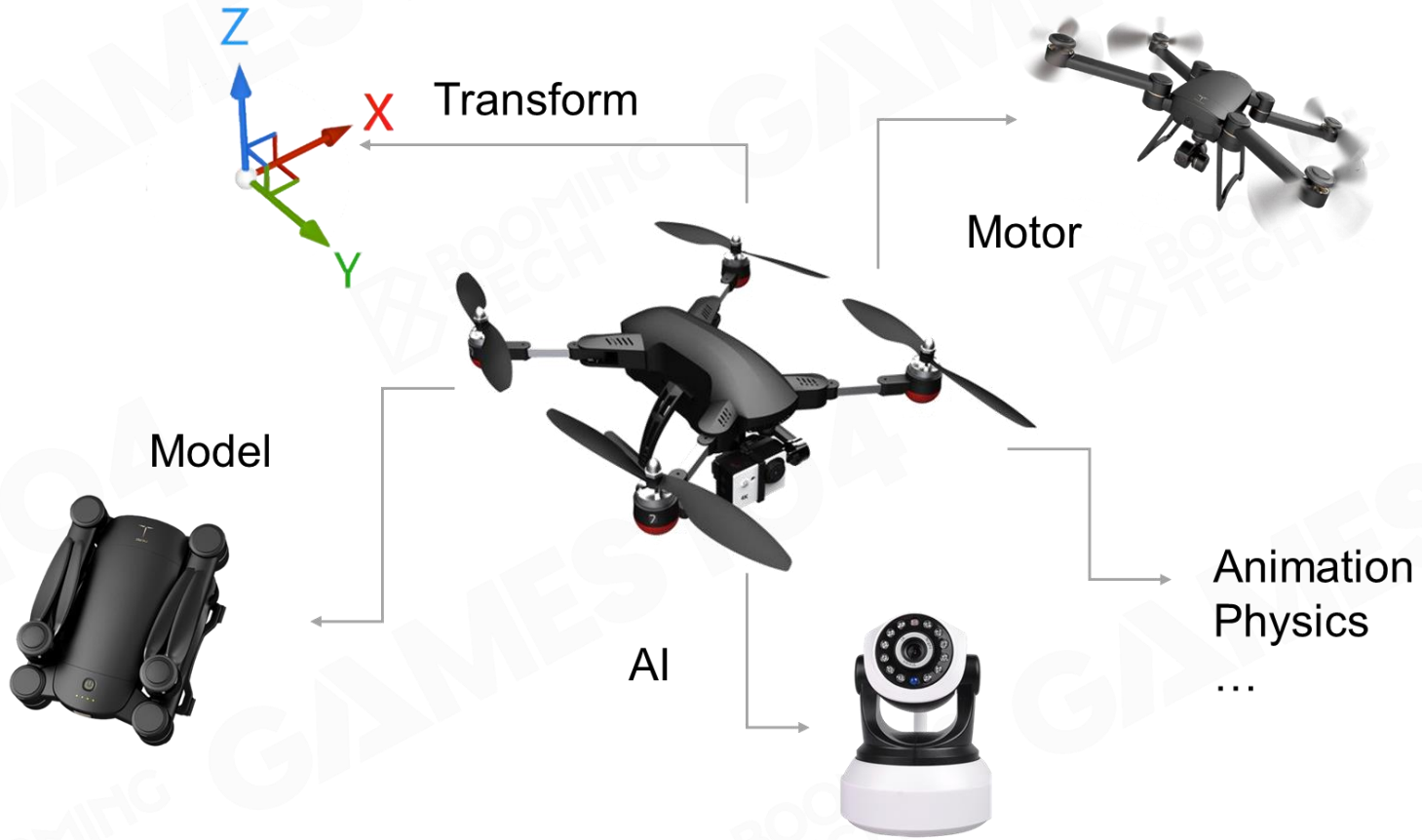
SOA

```
struct Particles{
    Vector3 position[N];
    Vector3 velocity[N];
    Color color[N];
    float age[N];
    //...
} Particles;
```



Entity Component System (ECS)

Recap: Component-based Design (1/2)



Recap: Component-based Design (2/2)

- Code example

Base class of component

```
class ComponentBase
{
    virtual void tick() = 0;
    ...
};
```

```
class GameObjectBase
{
    vector<ComponentBase*> components;
    virtual void tick();
    ...
};

class Drone:
    public GameObjectBase
{
    ...
};
```

```
class TransformComponent:
    public ComponentBase
{
    Vector3 position;
    ...
    void tick();
};
```


```
class ModelComponent:
    public ComponentBase
{
    Mesh mesh;
    ...
    void tick();
};
```

```
class MotorComponent:
    public ComponentBase
{
    float battery;
    void tick();
    void move();
    ...
};
```

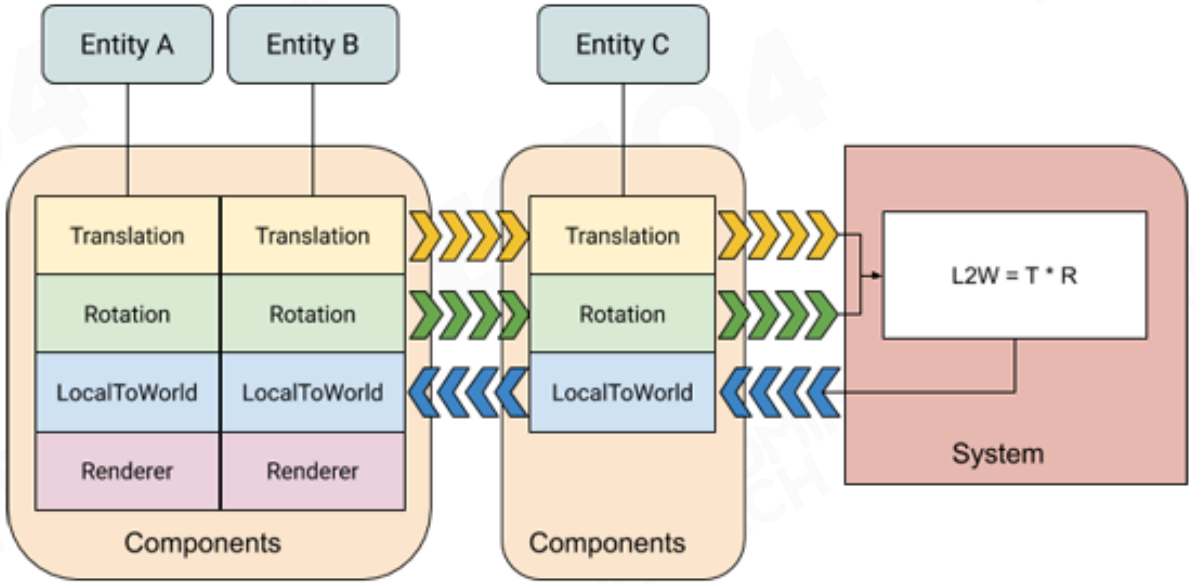
```
class AIComponent:
    public ComponentBase
{
    void tick();
    void scout();
    ...
};
```

Animation
Physics
…

Entity Component System (ECS)

A pattern to structure game code in a data-oriented way for maximum performance

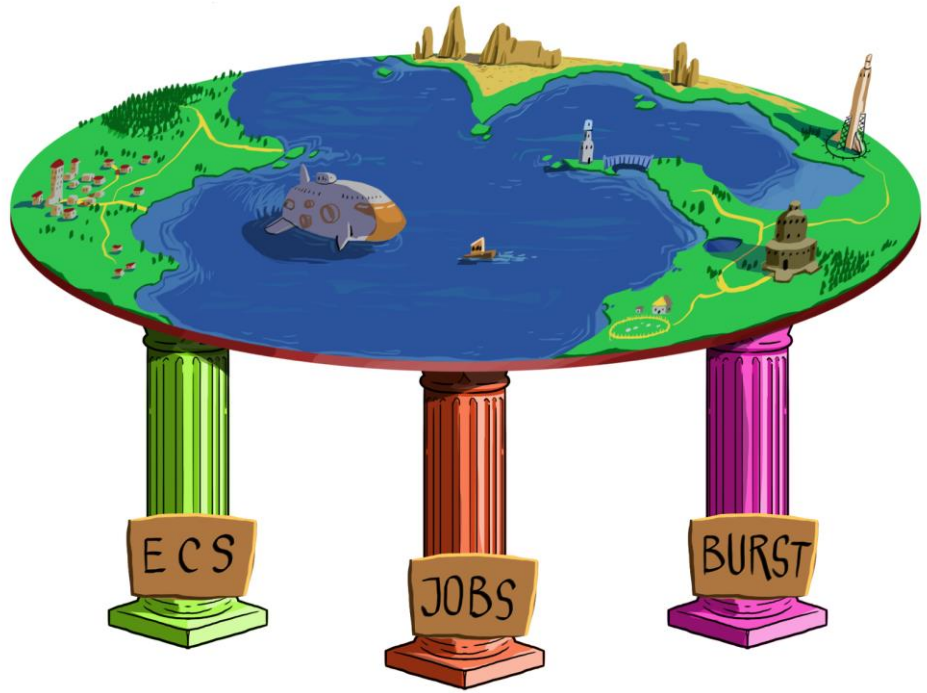
- **Entity**: an ID refer to a set of components
- **Component**: the data to be processed by systems, no logic at all
- **System**: where the logic happens, read/write component data



Unity Data-Oriented Tech Stack (DOTS)

A combination of technologies that work together to deliver a data-oriented approach to coding

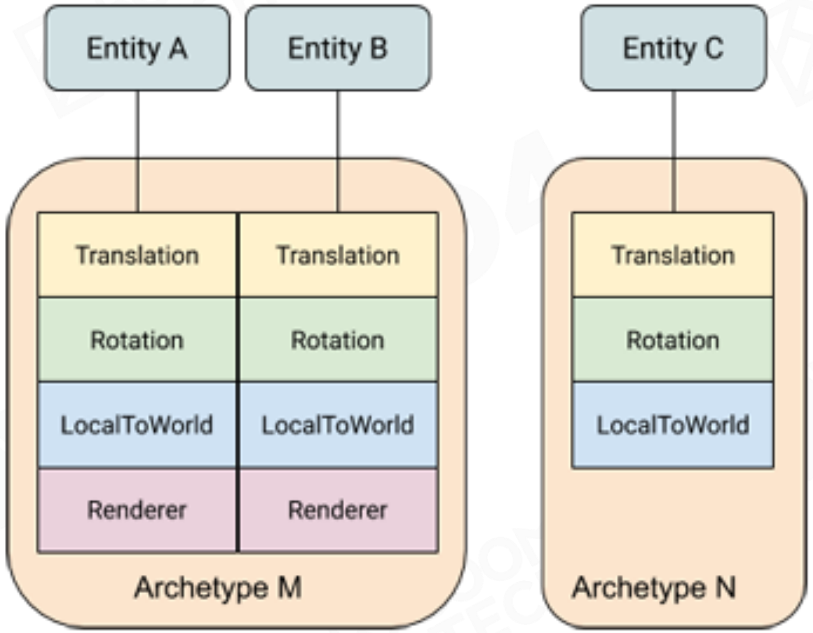
- The **Entity Component System (ECS)** provides data-oriented programming framework
- The **C# Job System** provides a simple method of generating multithreaded code
- The **Burst Compiler** generates fast and optimized native code



Unity ECS – Archetype

A specific combination of components

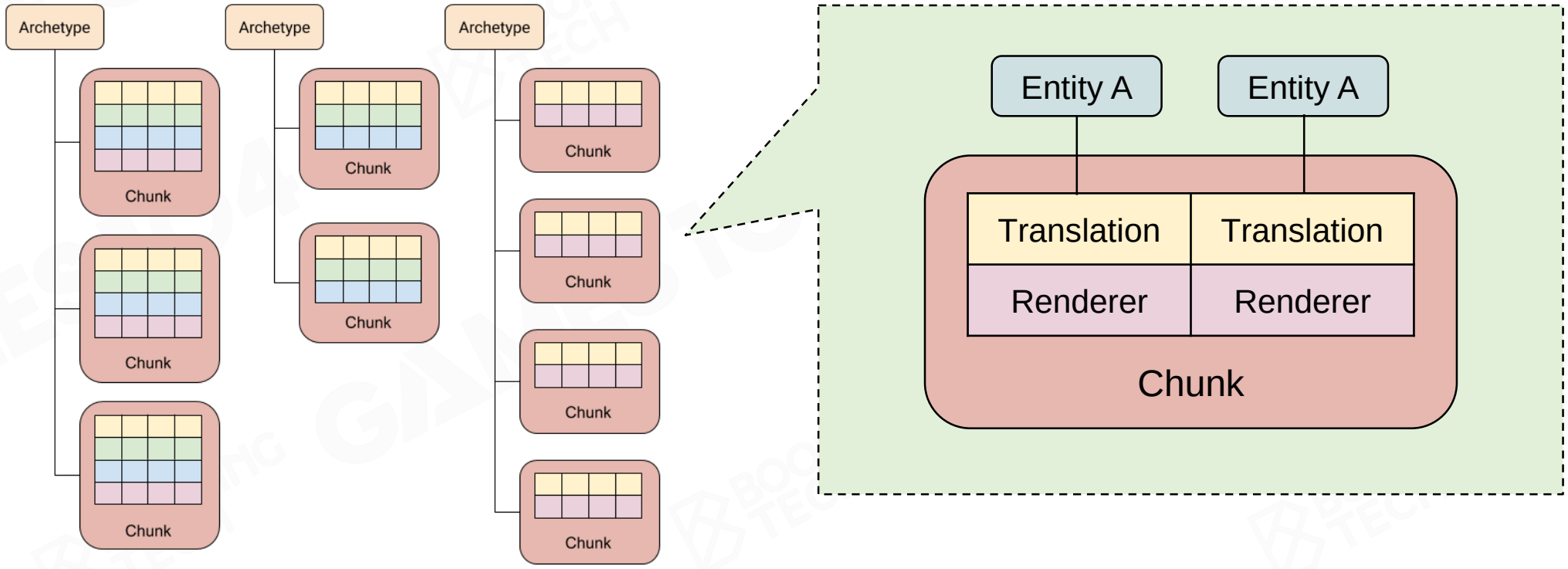
- Entities are grouped into archetypes



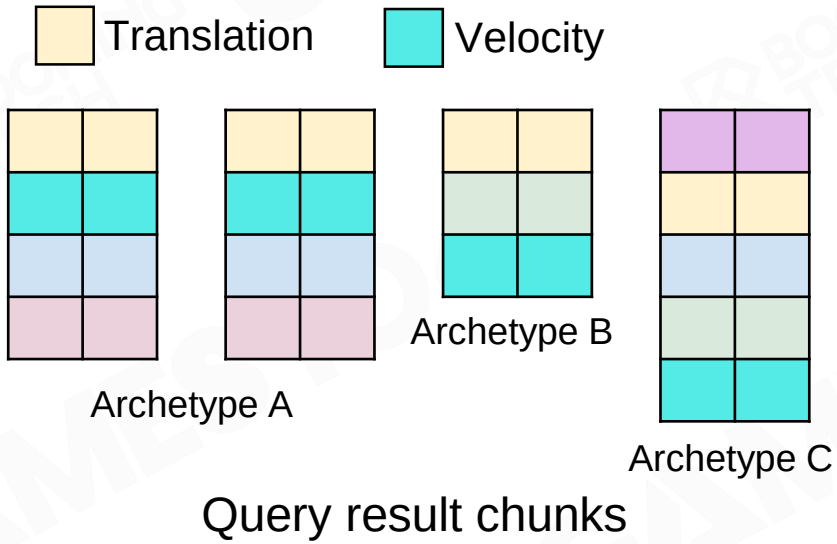
Unity ECS – Data Layout in Archetype

Same components in an archetype are packed tightly into chunks for cache friendliness

- A chunk is a block of memory with fixed size, i.e. 16KB



Unity ECS – System




```
public class MoveSystem : SystemBase
{
    protected override void OnUpdate()
    {
        // For each entity which has Translation and Velocity
        Entities.ForEach(
            // Write to Displacement (ref), read Velocity (in)
            (ref Translation trans, in Velocity velocity) =>
            {
                //Execute for each selected entity
                trans = new Translation()
                {
                    // dT is a captured variable
                    Value = trans.Value + velocity.Value * dT
                };
            }
        )
        .ScheduleParallel(); // Schedule as a parallel job
    }
}
```

Unity C# Job System

Make it easier for users to write correct multithreaded code

- A job is a small unit of work that performs a specific task
- Jobs can depend on other jobs to complete before they run

```
public struct FirstJob : IJob
{
    public void Execute()
    {
        ...
    }
}

public struct SecondJob : IJob
{
    public void Execute()
    {
        ...
    }
}
```

```
var first_job = new FirstJob();
var second_job = new SecondJob();

// execute first_job
var first_job_handle = first_job.Schedule();

// second_job depends on first_job to complete
second_job.Schedule(first_job_handle);
```

Unity C# Job System – Native Container

A type of shared memory that can be accessed inside jobs

- Job cannot output result without native container (all data is a copy)
- Native containers support all safety checks
- Native containers need to be disposed manually

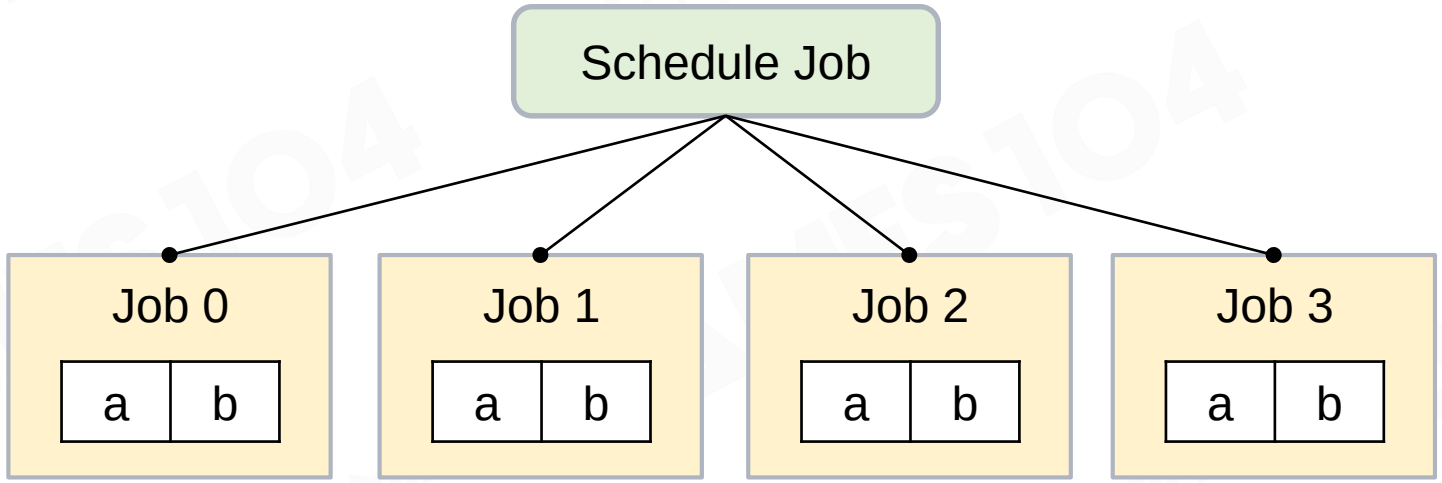
```
// Allocate one float with "TempJob" policy
// Allocator.Temp:       Fastest allocation, lifespan is 1 frame or fewer
// Allocator.TempJob:    Slower than Temp, lifespan is 4 frames
// Allocator.Persistent: Slowest allocation, can last as long as needed
NativeArray<float> a = new NativeArray<float>(1, Allocator.TempJob);

...

// Need to dispose manually for unmanaged memory
a.Dispose();
```

Unity C# Job System – Safety System

Support safety checks (out of bounds checks, deallocation checks, race condition checks) for jobs

- Send each job a copy of data it needs to operate on to eliminate the race condition
  - Job can only access blittable data types (reference is invalid)

```
public struct Job : IJob
{
    public float a;
    public float b;

    public void Execute()
    {
        ...
    }
}
```



Each job has a copy of data

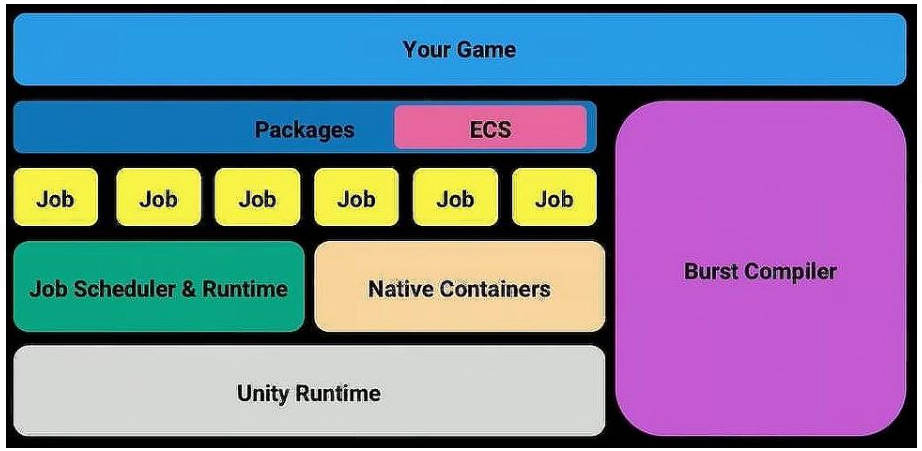
High-Performance C# and Burst Compiler

High-Performance C# (HPC#) is a subset of C#

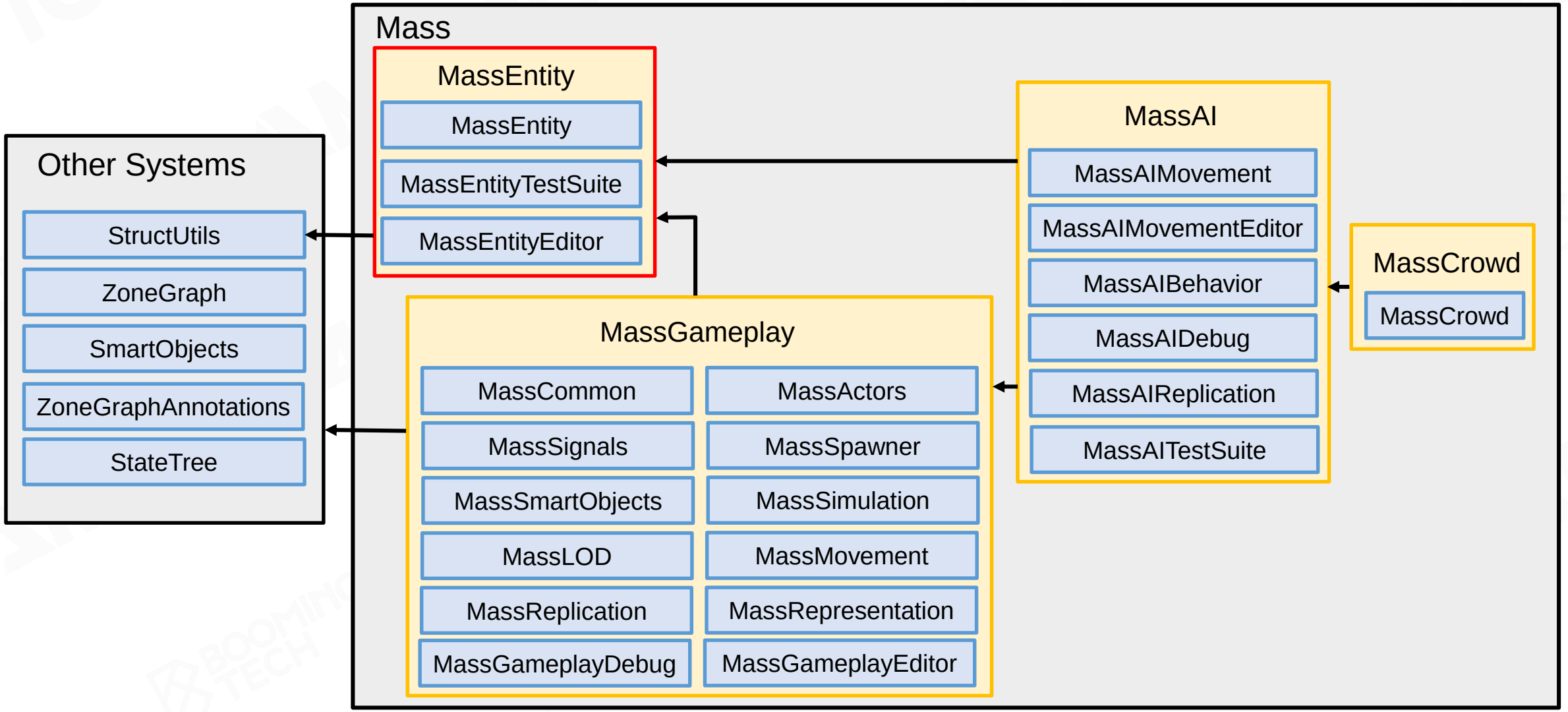
- Give up on most of the standard library (StringFormatter, List, Dictionary, and etc.)
- Disallow allocations, reflection, the garbage collector and virtual calls

Burst Compiler translates from IL/.NET bytecode to highly optimized native code using LLVM

- Generate expected machine code for specific platforms



Unreal Mass Framework



MassEntity – Entity

- FMassEntityHandle is **pure ID** as ECS **Entity**
- Index indicates the index in Entities array in FMassEntityManager
- SerialNumber as salt to Index
  - Release an old entity
  - Create a new entity with the same Index
  - SerialNumber is increased so the ID will be different

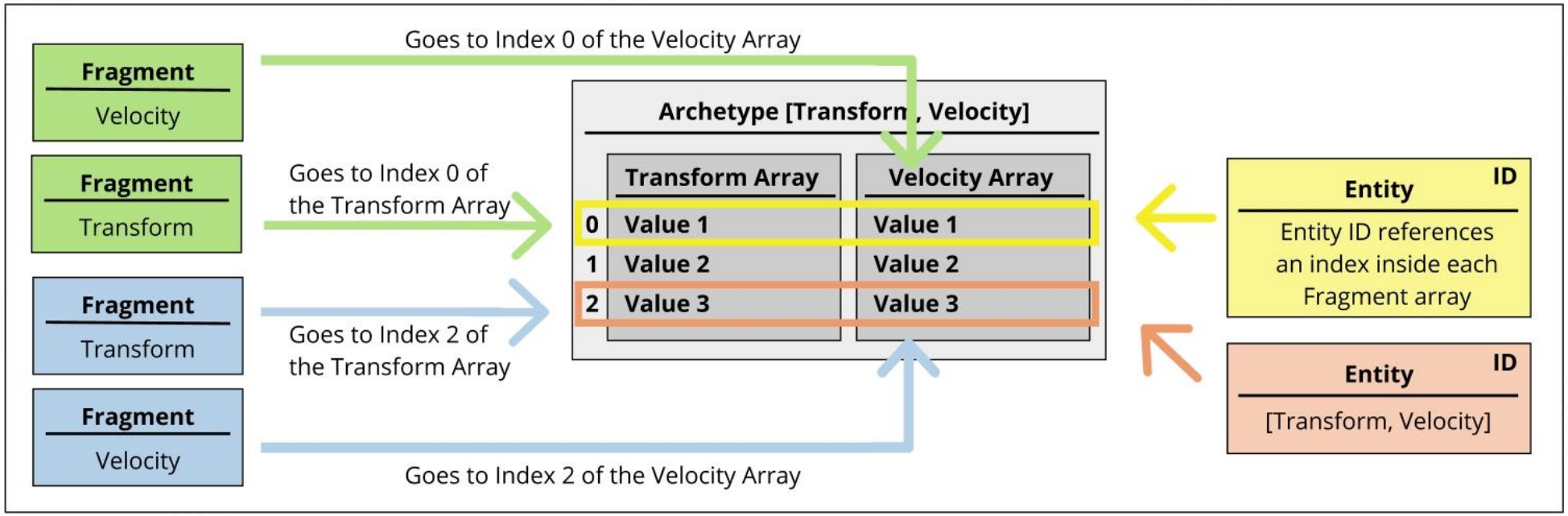
```
struct FMassEntityHand1e
{
    ...
    int32 Index = 0;
    int32 SerialNumber = 0;
    ...
}
```

```
struct MASSENTITY_API FMassEntityManager
{
    ...
    TChunkedArray<FEntityData> Entities;
    TArray<int32> EntityFreeIndexList;
    ...
}
```

MassEntity – Component

- Same as Unity, each type of entity has an Archetype
- **Fragments and tags** are **components** for entities
- Tags are constant Boolean components to filter unnecessary processing

```
struct FMassArchetypeCompositionDescriptor
{
    ...

    FMassFragmentBitSet Fragments;
    FMassTagBitSet Tags;
    FMassChunkFragmentBitSet ChunkFragments;
    FMassSharedFragmentBitSet SharedFragments;
}
```



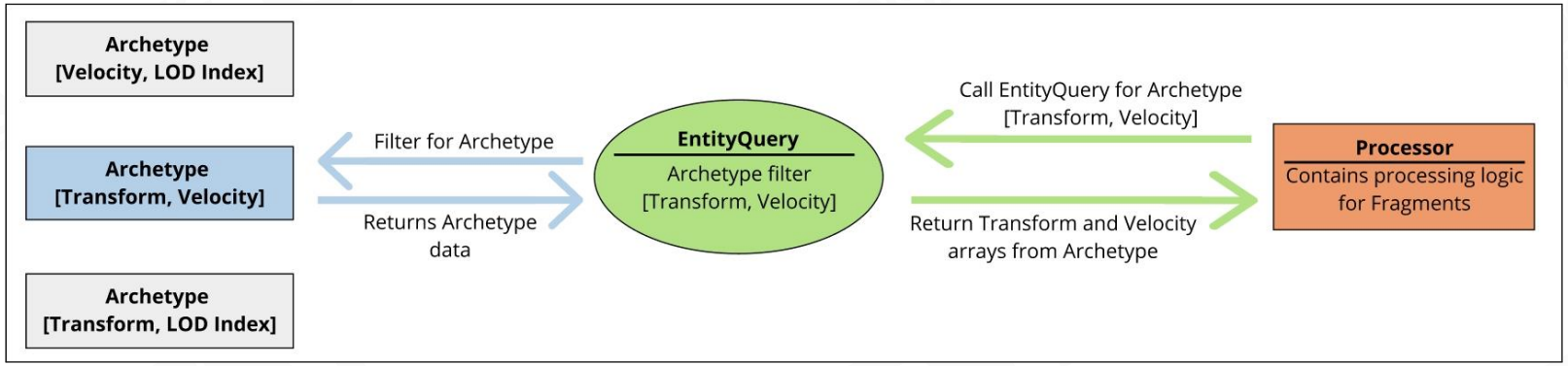
MassEntity – Systems

- ECS **Systems** in MassEntity are **Processors** derived from UMassProcessor
- Two important interface: **ConfigureQueries()** and **Execute(...)**

```
class MASSENTITY_API UMassProcessor : public UObject
{
    ...
protected:
    virtual void ConfigureQueries() PURE_VIRTUAL(UMassProcessor::ConfigureQueries);
    virtual void PostInitProperties() override;
    virtual void Execute(
        FMassEntityManager& EntityManager,
        FMassExecutionContext& Context) PURE_VIRTUAL(UMassProcessor::Execute);

    ...
}
```

MassEntity – Fragment Query

- **Interface ConfigureQueries()** runs when the processor is initialized
- Use FMassEntityQuery to filter archetypes of entities meeting systems requirements
- FMassEntityQuery caches filtered archetypes to accelerate future executions




```
void UMassApplyMovementProcessor::ConfigureQueries()
{
    EntityQuery.AddRequirement<FMassVelocityFragment>(EMassFragmentAccess::ReadWrite);
    EntityQuery.AddRequirement<FTransformFragment>(EMassFragmentAccess::ReadWrite);
    EntityQuery.AddRequirement<FMassForceFragment>(EMassFragmentAccess::ReadWrite);
    EntityQuery.AddTagRequirement<FMassOffLODTag>(EMassFragmentPresence::None);
    EntityQuery.AddConstSharedRequirement<FMassMovementParameters>(EMassFragmentPresence::All);
}
```

MassEntity – Execute

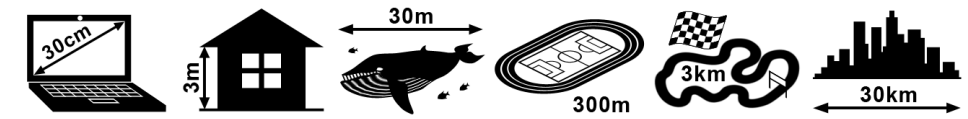
```
void UMassApplyMovementProcessor::Execute(FMassEntityManager& EntityManager,
                                          FMassExecutionContext& Context)
{
    // Clamp max delta time to avoid force explosion on large time steps (i.e. during initialization).
    const float DeltaTime = FMath::Min(0.1f, Context.GetDeltaTimeSeconds());
    EntityQuery.ForEachEntityChunk(EntityManager, Context, [this, DeltaTime](FMassExecutionContext& Context)
    {
        const int32 NumEntities = Context.GetNumEntities();
        const TArrayView<FTransformFragment> LocationList = Context.GetMutableFragmentView<FTransformFragment>();
        const TArrayView<FMassForceFragment> ForceList = Context.GetMutableFragmentView<FMassForceFragment>();
        const TArrayView<FMassVelocityFragment> VelocityList = Context.GetMutableFragmentView<FMassVelocityFragment>();
        for (int32 EntityIndex = 0; EntityIndex < NumEntities; ++EntityIndex)
        {
            FMassForceFragment& Force = ForceList[EntityIndex];
            FMassVelocityFragment& Velocity = VelocityList[EntityIndex];
            FTransform& CurrentTransform = LocationList[EntityIndex].GetMutableTransform();
            // Update velocity from steering forces.
            Velocity.Value += Force.Value * DeltaTime;
            ...
            FVector CurrentLocation = CurrentTransform.GetLocation();
            CurrentLocation += Velocity.Value * DeltaTime;
            CurrentTransform.SetTranslation(CurrentLocation);
        }
    });
}
```

Conclusions

Everything You Need Know About Performance

Not all CPU operations are created equal

Operation	Cost in CPU Cycles
“Simple” register-register op (ADD,OR,etc.)	<1
Memory write	~1
Bypass delay: switch between integer and floating-point units	0-3
“Right” branch of “if”	1-2
Floating-point/vector addition	1-3
Multiplication (integer/float/vector)	1-7
Return error and check	1-7
L1 read	3-4
TLB miss	7-21
L2 read	10-12
“Wrong” branch of “if” (branch misprediction)	10-20
Floating-point division	10-40
128-bit vector division	10-70
Atomics/CAS	15-30
C function direct call	15-30
Integer division	15-40
C function indirect call	20-50
C++ virtual function call	30-60
L3 read	30-70
Main RAM read	100-150
NUMA: different-socket atomics/CAS (guesstimate)	100-300
NUMA: different-socket L3 read	100-300
Allocation+deallocation pair (small objects)	200-500
NUMA: different-socket main RAM read	300-500
Kernel call	1000-1500
Thread context switch (direct costs)	2000
C++ Exception thrown+caught	5000-10000
Thread context switch (total costs, including cache invalidation)	10000 - 1 million



References

Cache

- Entity Component Systems & Data Oriented Design, Unity Training Academy 2018-2019, #3
  https://aras-p.info/texts/files/2018Academy%20-%20ECS-DoD.pdf
- Computer Architecture: A Quantitative Approach 5th Edition by John L. Hennessy , David A. Patterson
- What is the bandwith speed of L1,L2 and L3 Cache
  https://linustechtips.com/topic/34636-what-is-the-bandwith-speed-of-l1l2-and-l3-cache/
- Intel Core i9-9900K CPU Review: More Cores, Speed, and Higher Price
  https://www.overclockers.com/intel-core-i9-9900k-cpu-review-more-cores-speed-and-higher-price/
- Wikipedia - Cache replacement policies
  https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_(LRU)

Parallel Programming (1/3)

- Operating System Basics (Brian Will) https://linuxwheel.com/operating-system-basics-brian-will/
- Parallel computing via multicore computers allow high processing capacity https://www.teldat.com/blog/parallel-computing-bit-instruction-task-level-parallelism-multicore-computers/
- Internals of a Thread Pool https://salonegupta.wordpress.com/2017/12/28/internals-of-a-java-thread-pool/
- CPP Reference - Atomic https://en.cppreference.com/w/cpp/atomic
- TBB Tutorial https://www.inf.ed.ac.uk/teaching/courses/ppls/TBBtutorial.pdf

Parallel Programming (2/3)

- Parallel Programming Models and Paradigms
  http://www.cse.hcmut.edu.vn/~hungnq/courses/pp/backup.2/thamkhao/Parallel%20Programming%20Paradigms.pdf
- Parallel Paradigms and Parallel Algorithms https://pdc-support.github.io/introduction-to-mpi/05-parallel-paradigms/index.html
- Developing Parallel Programs - A Discussion of Popular Models
  https://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/oss-parallel-programs-170709.pdf
- Modern Fortran: Building efficient parallel applications MEAP V13
  https://livebook.manning.com/book/modern-fortran/welcome/v-13/

Parallel Programming (3/3)

- Priority Inversion http://www.embeddedlinux.org.cn/rtconforembsys/5107final/LiB0101.html
- What is a Thread in OS and what are the differences between a Process and a Thread? https://afteracademy.com/blog/what-is-a-thread-in-os-and-what-are-the-differences-between-a-process-and-a-thread
- Understanding operating systems https://www.uow.edu.au/student/learning-co-op/technology-and-software/operating-systems/

Parallel Frameworks in Game Engine (1/2)

- GDC2015 - Parallelizing the Naughty Dog Engine Using Fibers https://www.gdcvault.com/play/1022186/Parallelizing-the-Naughty-Dog-Engine
- GCAP 2016: Parallel Game Engine Design - Brooke Hodgman https://www.youtube.com/watch?v=JpmK0zu4Mts
- Java - Thread Pools https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/thread-pools.html
- C++20: Building a Thread-Pool With Coroutines https://blog.eiler.eu/posts/20210512/
- Processes, threads, and coroutines https://subscription.packtpub.com/book/programming/9781788627160/1/ch01lvl1sec02/processes-threads-and-coroutines

Parallel Frameworks in Game Engine (2/2)

- UE并发-TaskGraph的实现和用法 https://zhuanlan.zhihu.com/p/398843895?utm_medium=social&utm_oi=1447486643037528064&utm_psn=1546525648855732224&utm_source=ZHShareTargetIDMore
- Unreal Engine 5.0 Documentation - Tasks System https://docs.unrealengine.com/5.0/en-US/tasks-systems-in-unreal-engine/
- UE4/UE5的TaskGraph https://cloud.tencent.com/developer/article/1897046

DOP (1/3)

- Programming Paradigms – Paradigm Examples for Beginnershttps://www.freecodecamp.org/news/an-introduction-to-programming-paradigms/#what-is-a-programming-paradigm
- GDC'cn 为实现极限性能的面向数据编程范式 叶劲峰https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdcchina14/presentations/833779_MiloYip_ADataOrientedCN.pdf
- Timeline of Computer History https://www.computerhistory.org/timeline/computers/
- The Fetch and Execute Cycle: Machine Language https://math.hws.edu/javanotes-swing/c1/s1.html
- Wikipedia-Single instruction, multiple data

DOP (2/3)

- Linked List (Data Structure) https://devopedia.org/linked-list-data-structure
- Sekiro: Shadows Die Twice - All Bosses [No Damage] https://www.youtube.com/watch?v=KPAvM2hcSH8
- Ori 2 - Boss - Mora (Giant Spider) - Hard Difficulty https://www.youtube.com/watch?v=tuhrtBRLQPw
- The Greatest Frame Loss of All Time https://www.youtube.com/watch?v=4efRYXuhVTA
- Monster Hunter World | Great Sword Tutorial https://www.youtube.com/watch?v=X2vr8M3lQ88

DOP (3/3)

- Data-Oriented Design, Fabian R, CRC Press, 2018.
  https://www.dataorienteddesign.com/dodbook.pdf
- OOP Is Dead, Long Live Data-oriented Design. Nikolov S, Coherent Labs. CppCon 2018.
  https://www.bilibili.com/video/BV1kW41117uw?p=66&vd_source=f12a5db552661d28e8507875c37983cd
- Data-Oriented Design and C++, Acton M. Insomniac Games. CppCon 2014.
  https://www.youtube.com/watch?v=rX0ItVEVjHc
- Data-Oriented Design Resources: https://github.com/dbartolini/data-oriented-design

Unity DOTS (1/2)

- Getting Started with Unity DOTS https://nikolayk.medium.com/getting-started-with-unity-dots-part-1-ecs-7f963777db8e
- Unity Manual - ParallelFor Jobs https://docs.unity3d.com/Manual/JobSystemParallelForJobs.html
- Unity Learn - What is DOTS and why is it important https://learn.unity.com/tutorial/what-is-dots-and-why-is-it-important#
- On DOTS: Entity Component System https://blog.unity.com/technology/on-dots-entity-component-system
- Unite Los Angeles 2018 Keynote https://www.youtube.com/watch?v=alZ6wmwvck0&t=6434s

Unity DOTS (2/2)

- Building a Data-Oriented Future - Mike Acton
  https://www.youtube.com/watch?v=u8B3j8rqYMw
- Getting started with Unity DOTS — Part 2: C# Job System
  https://nikolayk.medium.com/getting-started-with-unity-dots-part-2-c-job-system-6f316aa05437

Unreal Engine Mass Architecture

- UE5 MassEntity Documentation, https://docs.unrealengine.com/5.0/en-US/overview-of-mass-entity-in-unreal-engine/
- UE5的ECS：MASS框架(一), quabqi, 2022, https://zhuanlan.zhihu.com/p/441773595
- UE5的ECS：MASS框架(二), quabqi, 2022, https://zhuanlan.zhihu.com/p/446937133
- UE5的ECS：MASS框架(三), quabqi, 2022, https://zhuanlan.zhihu.com/p/477803528

Multimedia Material List

- Sekiro: Shadows Die Twice - All Bosses [No Damage] https://www.youtube.com/watch?v=KPAvM2hcSH8
- Ori 2 - Boss - Mora (Giant Spider) - Hard Difficulty https://www.youtube.com/watch?v=tuhrtBRLQPw
- Monster Hunter World | Great Sword Tutorial https://www.youtube.com/watch?v=X2vr8M3lQ88
- Review in Progress: Battlefield 2042 https://www.destructoid.com/review-in-progress-battlefield-2042-ps5-version/
- Infographics: Operation Costs in CPU Clock Cycles, http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

Lecture 20 Contributors

- 一将
- 蓑笠翁
- 大喷
- 果蝇
- Olorin
- 灰灰
- 喵小君

Q&A

Enjoy ;)
Coding

Course Wechat

Please follow us for further information