Combining Co-Routines and Functions into a Job System

HELMUT HLAVACS

Cppcon
The C++ Conference
2021
October 24-29

About Myself

- Professor for Computer Science
- University of Vienna, Austria: founded 1365, >90000 students
- Entertainment Computing Research Group
  - **Efficiency and performance of game engines,** AI, networking, VR …
- Teaching: 3D Graphics, AI, Physics for games, Game Streaming, …
- **IFIP** (International Federation for Information Processing)
  Technical Committee 14 *Entertainment Computing*

Creating Game Engines with C++

- **Vienna Game Job System +**
- Graphics API Abstraction Layer +
- Vienna Entity Component System + Vienna Type List Library
- Vienna Physics Engine +
- Vienna Game AI Engine +
- GUI

- = Vienna Vulkan Game Engine 2.0

https://github.com/hlavacs

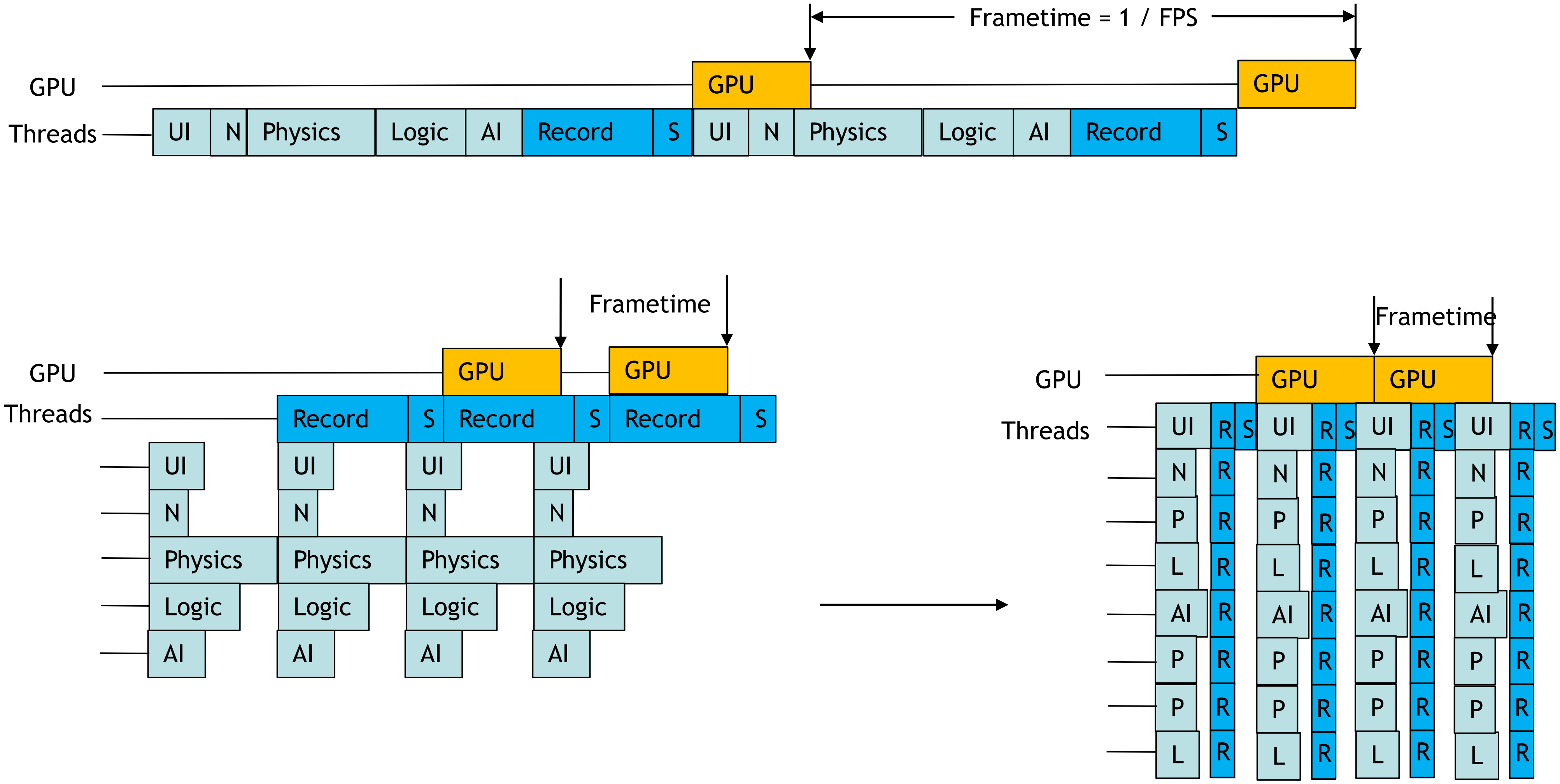
The Game Loop

```
auto prev = high_resolution_clock::now();

while( !finished() ) {

        auto now = high_resolution_clock::now();

        duration<double> delta_t = duration_cast<duration<double>>(now - prev);

        prev = now;

        window.tick(delta_t.count());

        network.tick(delta_t.count());

        physics.tick(delta_t.count());   //https://gafferongames.com/post/fix_your_timestep/

        game_logic.tick(delta_t.count());

        AI.tick(delta_t.count());

        //…

        prepare_render_next_frame();  //get idle frame buffer and record command buffers for it

        submit_for_render();          //submit command buffers

}
```

Modern Multicore CPUs

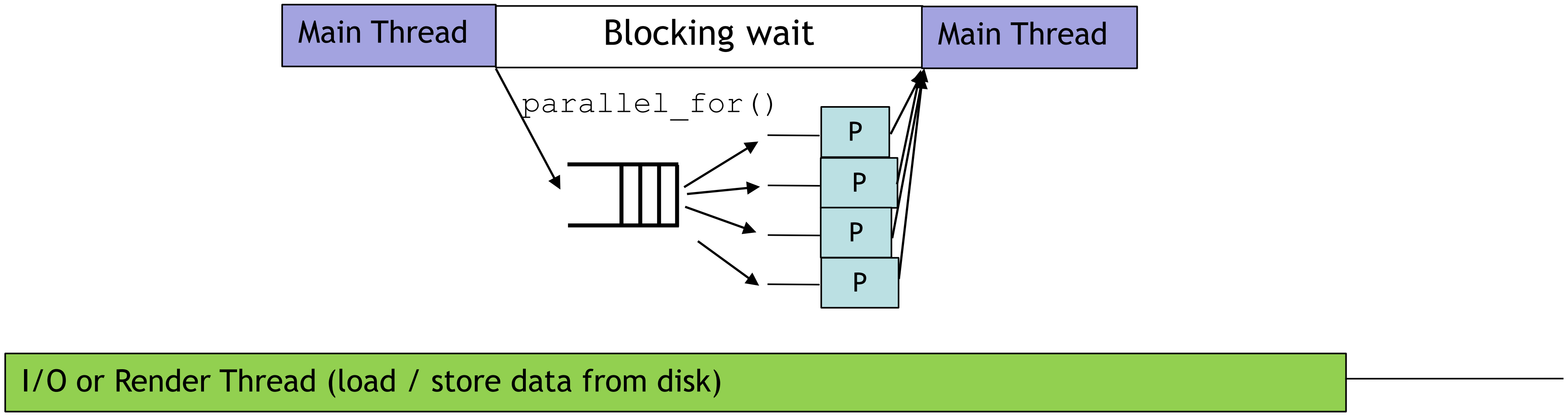
- N>1 independent *cores*
- Each core : 1 thread of execution (MIMD)
- Cores share main **memory**, can share **caches**
- Simultaneous multithreading (x86 / x64) -> 2N **virtual** cores
- Query number of cores: `std::thread::hardware_concurrency()`
- AMD : Ryzen: 2-64, Epyc 4-64
- Intel: Core i9: 6-18, Xeon: 4-56
- Apple: M1: 4+4

Reducing the Frametime

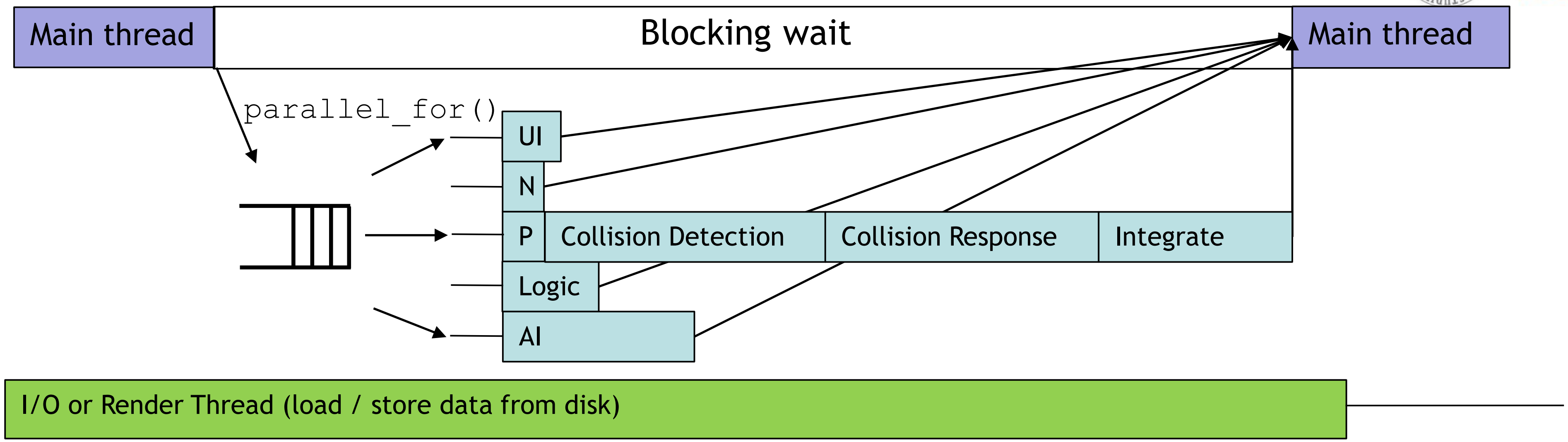


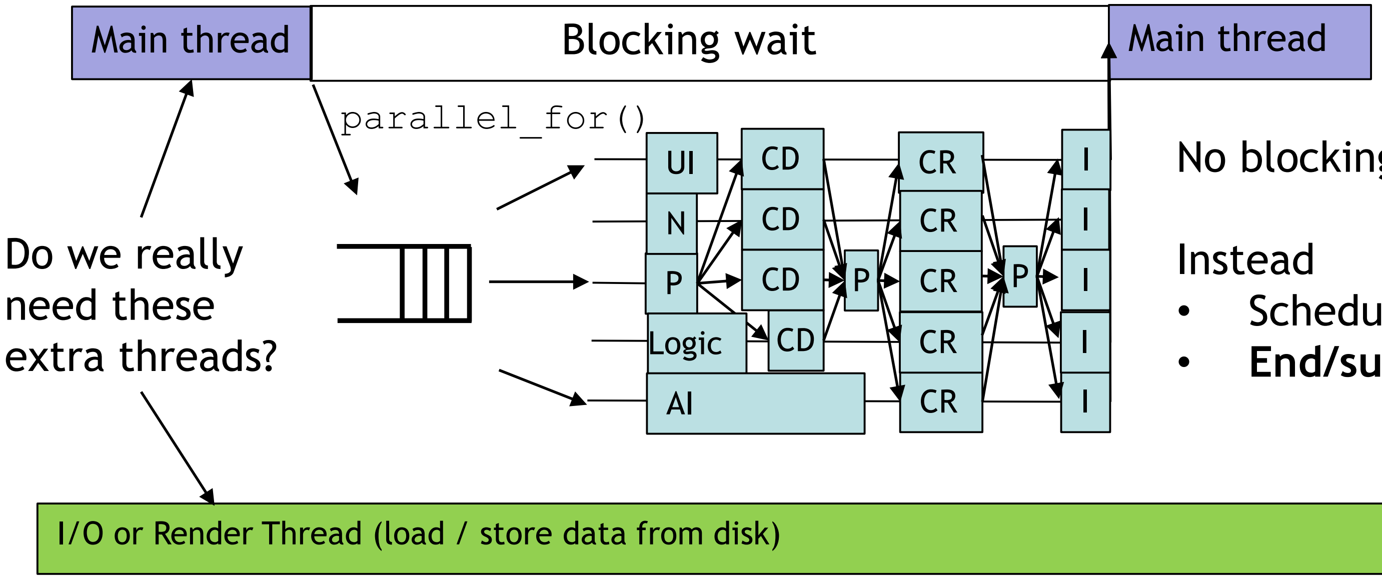
Job Systems and Thread Pools

- Starting threads is expensive -> **Thread Pool**
- **Job System** = Thread Pool + **Job Queues**
- **Main Thread calls** `parallel_for()`
  - Put jobs into queue
  - Threads take jobs out of queue



Main Thread + Job System + I/O Threads





No blocking wait!

Instead

- Schedule a **continuation**
- **End/suspend** caller

Do we really need extra threads? No we Don´t!

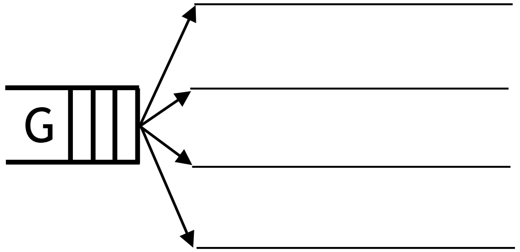
“Fun fact: Doom Eternal does not have a main or render thread. It‘s all jobs with one worker thread per core.“

Axel Gneiting, ID Software, March 21 2020

Further Improvements

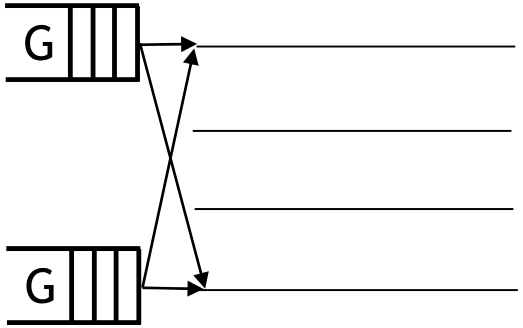
Thread-Pool only

- No thread outside the thread pool active



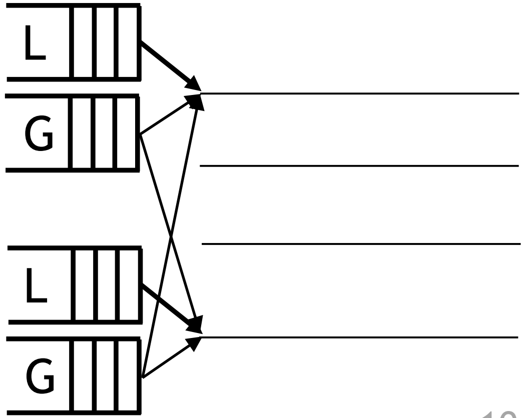
Work stealing: Each thread has its **own** (globally visible) **job queue**

- Steal jobs from other (global) queues -> **load balancing**



Locally and **Globally visible Queues**

- Local and a global (default) queue
- **Steal** from **global** queues



Tagged Scheduling

1	2

```
//Phase 1

schedule(f());

schedule(g(), tag_t{2});
```

```
//Phase 2

schedule(tag_t{2}); //schedule g()
```

The Vienna Game Job System (VGJS)

- Experimental job system for teaching and research,…
- https://github.com/hlavacs/ViennaGameJobSystem
- Thread Pool only, main thread can enter thread pool as worker, include file only
- Work stealing, 1 local and 1 global queue per thread, **tagged** scheduling
- Allocate from heap or memory resource
- Log performance and visualize in Google Chrome *chrome://tracing/*
- Scheduling jobs
  - `schedule(…)`
  - `continuation(…)`
  - `co_await …`

VGJS Thread Loop

```
void thread_task() noexcept {
   //initialize…
   while (!m_terminate) {  //Run until the job system is terminated
       m_current_job = m_local_queues[myidx.value].pop();                      //get a job from local queue
           if (m_current_job == nullptr) {
               m_current_job = m_global_queues[m_thread_index.value].pop();     //get a job from global queue
       }
       num_try = …;
       while (m_current_job == nullptr && --num_try >0) {
           if (++next >= m_thread_count) next = 0;
               m_current_job = m_global_queues[next].pop();   //try steal job from other global queue
               }
       }
       if (m_current_job != nullptr) {      //if I found a job
            (*m_current_job)();                 //run job
       } else {
           // sleep…                           //after some tries sleep
       }
   }
}
```

```
void f() {
   int var;
   //…
   return;
}
```

What can we schedule?

- **Normal Functions /class member functions**
  - **Lambdas**, **std::function<void(void)>**
  - **std::bind**
  - **void (*function)()**
  - struct **Function**{}: **std::function<void(void)>**, **thread**, **type and id** for logging
  - Tags

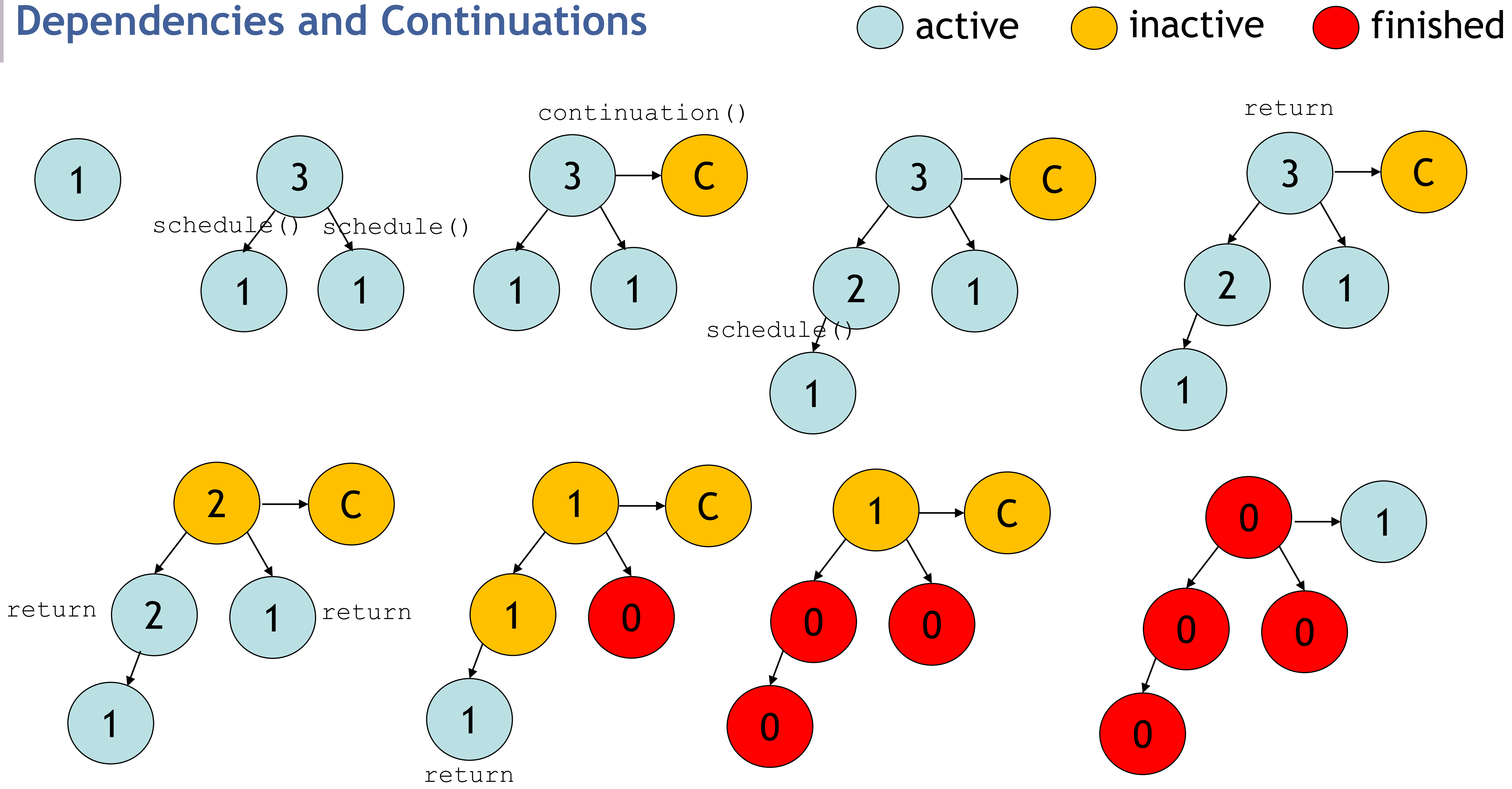
- **Coroutines** of type **Coro**<RETURNTYPE>
  - **Thread**, **type** and **id** with operator()

- **std::tuple** and **std::vector** containing an arbitrary number of them

Finishing and Continuations

- ***Finishing***: `return` + all children finished (counter == 0)
- *Notify* parent (if there is any)
- If ***continuation***, then schedule it

Dependencies and Continuations



Examples - Functions

```
void driver(int i) {
   //…
}
void end() {
   //schedule()/continuation()…
}
void test() {
   schedule([](){ driver(1); });                          //lambda

   std::function<void(void)> f1([](){ driver(2); }); //std::function
   schedule(f1);

   schedule(std::bind( driver, 3 ));                      //callable object

   schedule( Function{ [=]() {driver(3); },
       thread_index_t{ 0 }, thread_type_t{ 0 }, thread_id_t{ 1 } }); //Function

   continuation( end );                     //void(void)
}
```

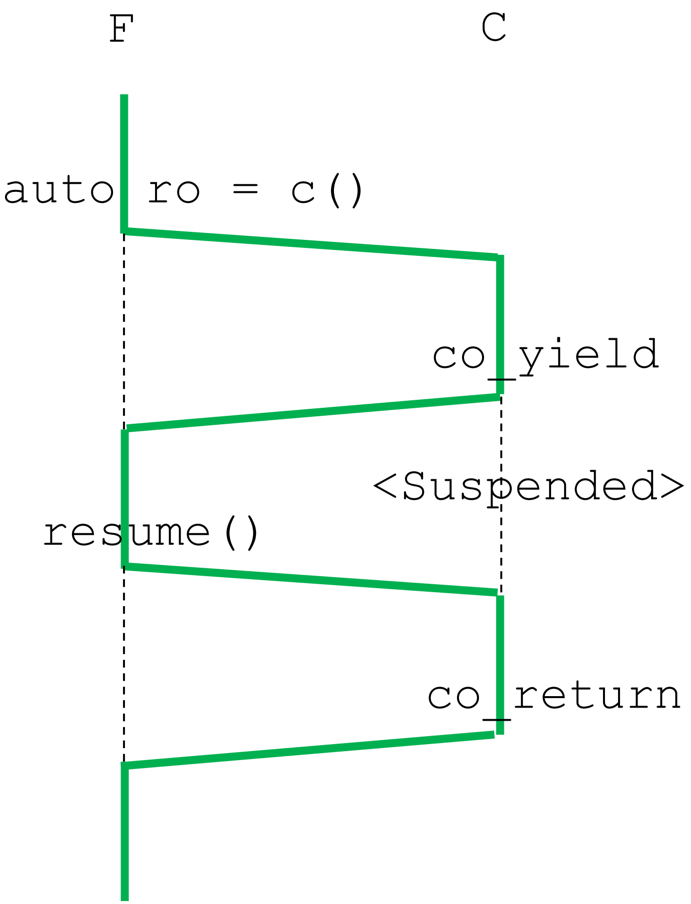
Coroutines

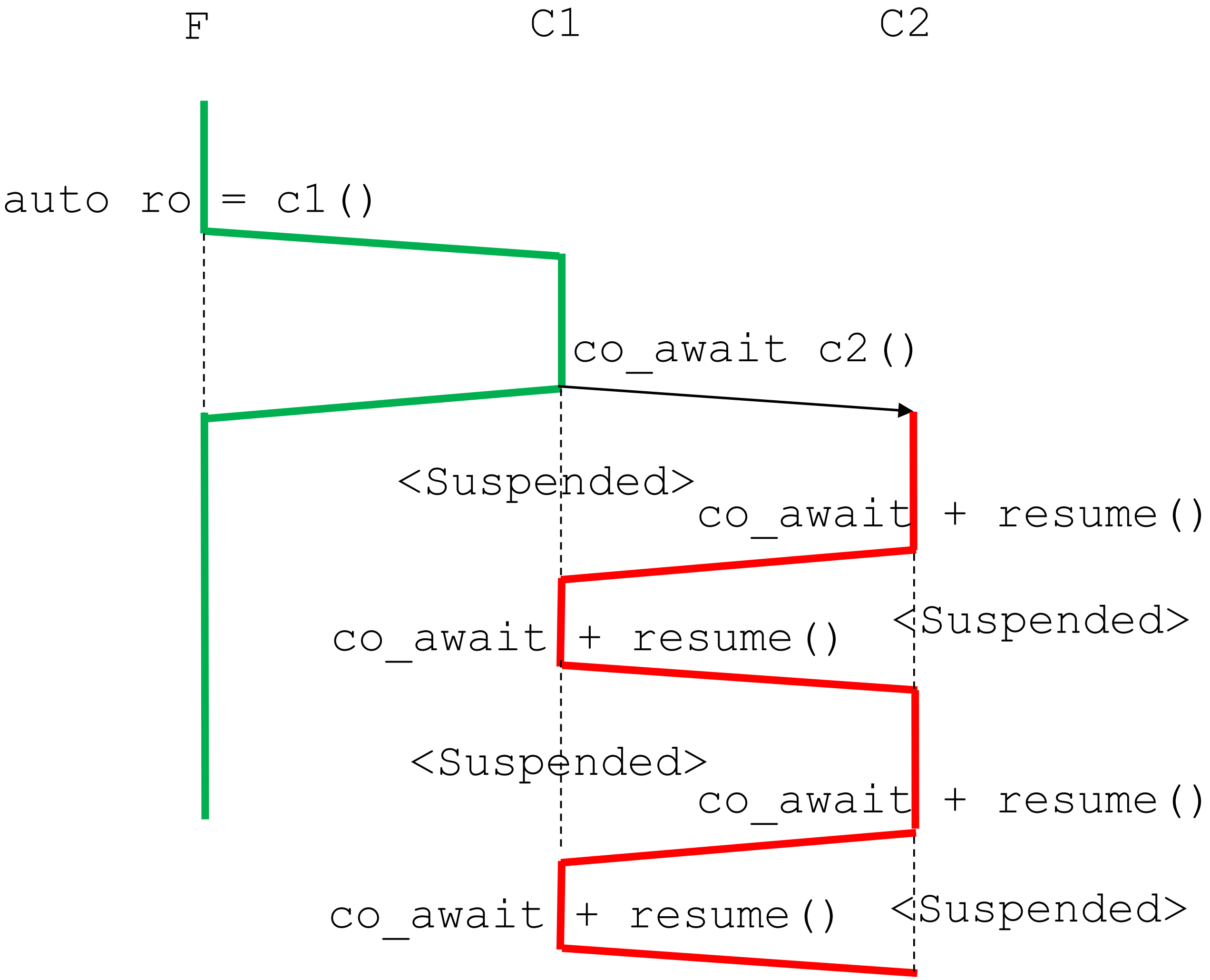
- Normal functions (synchronous)
  - Stack frame
  - **Gone** after `return`

- **Coroutines (asynchronous) can**
  - *Suspend* to wait for a result, *resume* with same state

  - **First time called:** allocate heap memory (*coroutine frame*)
  - **Suspend**: stack frame -> coroutine frame, **return** to caller/resumer
  - **Resume**: coroutine frame -> stack frame, **resume**
  - **Destroy**: deallocate coroutine frame

Coroutine Call Examples





Coroutines in VGJS

- Coroutines are created by **calling** them

```
Coro<float> retObj = driver(13); //create coro fr, return ret obj
```

- Schedule into VGJS queue
  - `schedule(retObj);            //from function`
  - `continuation(retObj);        //from function`
  - `co_await retObj;             //from coroutine`
  - `co_await parallel( retObj, Coro_vector, std_f1, …); //from coro`

- Thread grabs and *resumes* it

Coroutines in VGJS - Example

```
void test() {
    Coro<float> retObj = driver(13);
    schedule(retObj); //put promise into a VGJS queue, a thread will grab it in its loop
    if(retObj.ready()) std::cout << "Result " << retObj.get() << std::endl;
    return;
}

Coro<float> driver(int i) {
    int res = co_await coroTest(i);          //or co_await parallel(a(), vector, …)
    //resume here, a coroutine is its own continuation!
    co_return 0.0f;
}

Coro<int> coroTest(int i) {
    co_await thread_index_t{ 0 };
    co_yield 10;
    co_return 0;
}
```

VGJS Coroutines

```
void test() {
    Coro<float> retObj = driver(13);
    schedule(retObj);
}
```

Coroutine Frame
Suspend point
Function parameters
Stack frame
Coro_promise<float> promise;
promise.get_return_object()

Return Object Coro<T>
```
using promise_type =
    Coro_promise<T>;
```

Coro_promise<T>

```
Coro<float> driver(int i) {
  co_await promise.initial_suspend();
  try {
    co_return 0.0f;
  }
  catch (...) {
    promise.unhandled_exception();
  }
FinalSuspend:
  co_await promise.final_suspend();
}
```

```
Coro<float> driver(int i) {
  Coro<int> retObj = coroTest(i);
  int res = co_await retObj;
  Coro<int> retObj = coroTest(i);
  int res = co_await retObj;
  co_return 0.0f;
}
```

Awaiter

Awaiter

Awaiter

Coro_promise<T>

```
Coro<int> coroTest(int i) {
    co_await thread_index_t{ 0 };
    co_yield 10;
    co_return i;
}
```

Awaiter

Awaiter

Awaiter

Coroutine returning an int result

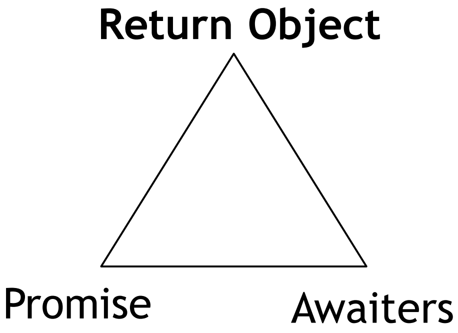
Main Classes to adapt Coroutine Behavior

Return Object for the caller

Promise:
Adapts the coroutine,
determines the **awaiters**

Awaiter:
Manages `co_await`
(suspend and resume)

Return Object

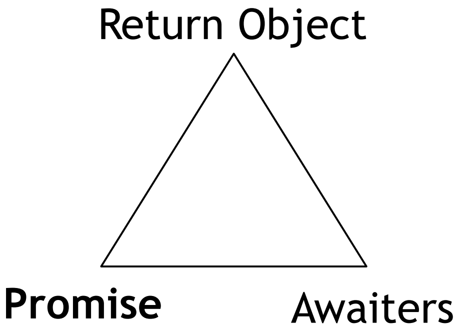


- **Specifies promise_type** (alternative: *coroutine_traits*)
- **Created** by the **promise** via `get_return_object()`
- Returned to the caller (at first suspension point / completion)

```
void test() {
  Coro<float> retObj = driver(13);
  //…
}
```

- VGJS return objects: test whether results **ready**, **get** results/promise ptr, **resume** coro, **set** thread index, type, id

Promises



- Alter **behavior** of coroutine through **API**

- Created by **first call** to **coroutine**, part of the **coroutine frame**

- **Return object** class **determines promise type** (`promise_type`), **but promise creates** the **return object**

- Destroyed when the coroutine frame gets destroyed

The Promise API in VGJS

```
template<typename T = void>
class Coro_promise : public Coro_promise_base {
   protected:
   //…

   public:
   Coro_promise() noexcept;
   suspend_always     initial_suspend() noexcept { return {}; }
   Coro<T>            get_return_object() noexcept;

   void               return_value(T t) noexcept;
   final_awaiter<T>   final_suspend() noexcept { return {}; };

   template<typename U>
   awaitable_tuple<T, U>      await_transform(U&& func) noexcept;
   template<typename... Ts>
   awaitable_tuple<T, Ts...>  await_transform(std::tuple<Ts...>&& tuple) noexcept;

   awaitable_resume_on<T>  await_transform(thread_index_t index) noexcept;
   awaitable_tag<T>        await_transform(tag_t tg) noexcept;

   yield_awaiter<T>        yield_value(T t) noexcept;
};
```

```
Coro<float> driver(int i) {
   co_await promise.initial_suspend();
   try {
      Coro<int> retObj = coroTest(i);
      int res = co_await retObj;
      co_return 0.0f;
   }
   catch (...) {
      promise.unhandled_exception();
   }
FinalSuspend:
   co_await promise.final_suspend();
}
```

//Coro<float> retObj = driver(13);

//co_return <value>

//co_await f();

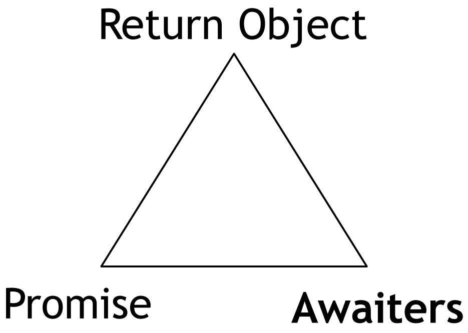
//ca parallel(…);

//co_await thread_index_t

//co_await tag_t

//co_yield <value>

Awaiters



- Manage `co_await` and `co_yield`

- **Default awaiters available** (e.g. for `initial_suspend()` etc.)
  - `suspend_always`
  - `suspend_never`

- Inherit default behavior

- Use directly, via **awaitable (operator co_await)** or **promise.await_transform(<expr>)** (co_await) https://lewissbaker.github.io/

What happens when you call `co_await <expr>`?

```
{
  auto&& awaiter = get_awaiter( promise, <expr> ); //promise.await_transform(<expr>)

  if (!awaiter.await_ready())   {                              //suspend?
    using handle_t = std::experimental::coroutine_handle<P>;

    <suspend-coroutine>          //suspend the coroutine

    if (awaiter.await_suspend(handle_t::from_promise(promise))) { //code after suspend
      <return-to-caller-or-resumer>
    }
    <resume-point>               //resume the coroutine
  }

  return awaiter.await_resume(); //after resume, return result
}
```

VGJS Awaiters

```
Coro<float> driver(int i) {
    co_await promise.initial_suspend();
    try {
        Coro<int> retObj = coroTest(i);
        int res = co_await retObj;
        co_return 0.0f;
    }
    catch (...) {
        promise.unhandled_exception();
    }
FinalSuspend:
    co_await promise.final_suspend();
}
```

- Initial_suspend (`suspend_always`)
- Final suspend
- `co_await parallel(…)    //wrapper for std::tuple<…>`
- `co_await thread_ID_t{…}`
- `co_await tag_t{…}`
- `co_yield <value>`

VGJS Awaiter for co_await parallel(…)

```
template<typename PT, typename... Ts>
struct awaitable_tuple : suspend_always {
   tag_t                  m_tag;        ///<The tag to schedule to
   std::tuple<Ts&&...>    m_tuple;      ///<tuple with all children to start
   std::size_t            m_number;     ///<total number of all new children to schedule

   template<typename U> size_t size(U& children) {}; //return number of children to schedule

   bool await_ready() noexcept {
      //count children for each tuple element, if no children then prevent suspension (true)
   }

   bool await_suspend(n_exp::coroutine_handle<Coro_promise<PT>> h) noexcept {
      //go through tuple elements and call schedule()
      return m_tag.value < 0;          //if tag value < 0 then schedule children (true)
   }

   auto await_resume() {
      //return the results from all coroutines in the tuple
   }

   awaitable_tuple(std::tuple<Ts&&...> tuple) noexcept :
      m_tag{}, m_number{0}, m_tuple(std::forward<std::tuple<Ts&&...>>(tuple)){};
};
```

VGJS Awaiter for `co_await thread_index_t{I}`

```
//co_await thread_index_t{0}

template<typename PT>
struct awaitable_resume_on : suspend_always {
   thread_index_t m_thread_index;                //the thread index to use

   //do not go on with suspension if the job is already on the right thread
   bool await_ready() noexcept {
      return (m_thread_index == JobSystem().get_thread_index());
   }

   void await_suspend(n_exp::coroutine_handle<Coro_promise<PT>> h) noexcept {
      h.promise().m_thread_index = m_thread_index;
      JobSystem().schedule_job(&h.promise());
   }

   awaitable_resume_on(thread_index_t index) noexcept : m_thread_index(index) {};
};
```

Coroutines can return Result Values

- **Coroutine** A coawaiting **coroutine** C
  - In **sync**, **return object** in A can destroy C

```
Coro<> A() {
   int res = co_await C();
}
```

- **Function** F scheduling **coroutine** C
  - F may return before C is finished -> C must destroyed **itself**
  - What if F tries to get the result but C is destroyed?
  - Store result in `std::shared_ptr<std::pair<bool, T>` shared by **return object** in F and **promise** of C

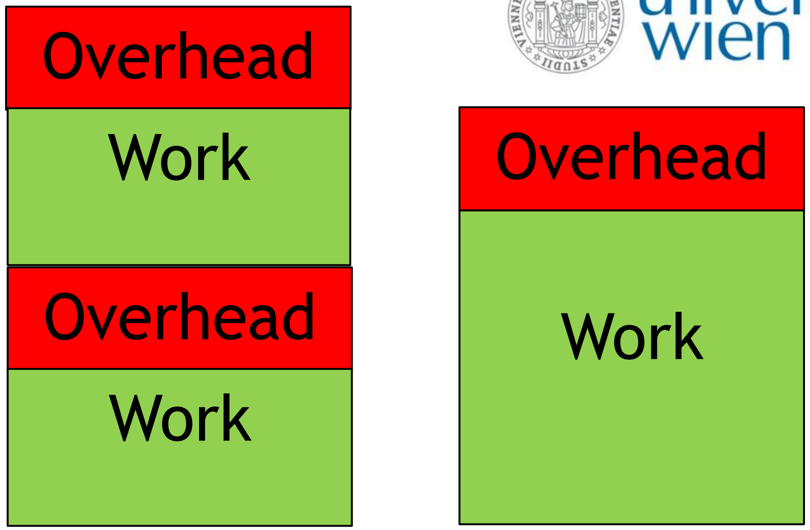
```
void F() {
   Coro<int> ro = C();
   schedule(ro);
   if(ro.ready()) {
       int res = ro.get()
   }
   return;
}
```

VGJS Final Awaiter of a Coroutine

```
Coro<float> driver(int i) {
    co_await promise.initial_suspend();
    try {
        Coro<int> retObj = coroTest(i);
        int res = co_await     retObj;
        co_return 0.0f;
    }
    catch (...) {
    promise.unhandled_exception();
    }
FinalSuspend:
    co_await promise.final_suspend();
}
```

```
//bool await_suspend()
//if true -> suspend (do not destroy the coroutine frame)
//if false -> do not suspend (destroy the coroutine frame)

template<typename U>
struct final_awaiter : public suspend_always {

  bool await_suspend(n_exp::coroutine_handle<Coro_promise<U>> h) noexcept {
     bool is_parent_function = …; //true if parent is a function

     //indicate parent that child has finished

     //if parent is coroutine -> suspend (return true)
     //if parent is function -> destroy (return false)
     return !is_parent_function;
  }
};
```

Performance Considerations



- Job Systems do have some **overhead**
- Scheduling, managing jobs, queues
- **Jobs should not be too small** (overhead would dominate)
- Increase **job size** to increase **effciciency**

Speed Up and Efficiency

Speed Up $S(n) := \frac{T_1}{T_n}$ Example: $T_1 = 100\mu s, T_4 = 50\mu s$ then $S(4) = 2$

Efficiency $E(n) := \frac{S(n)}{n}$ Example: $E(4) = 0.5$ or 50%

Questions

1. Minimum job size (CPU time) for 85/90/95% efficiency?

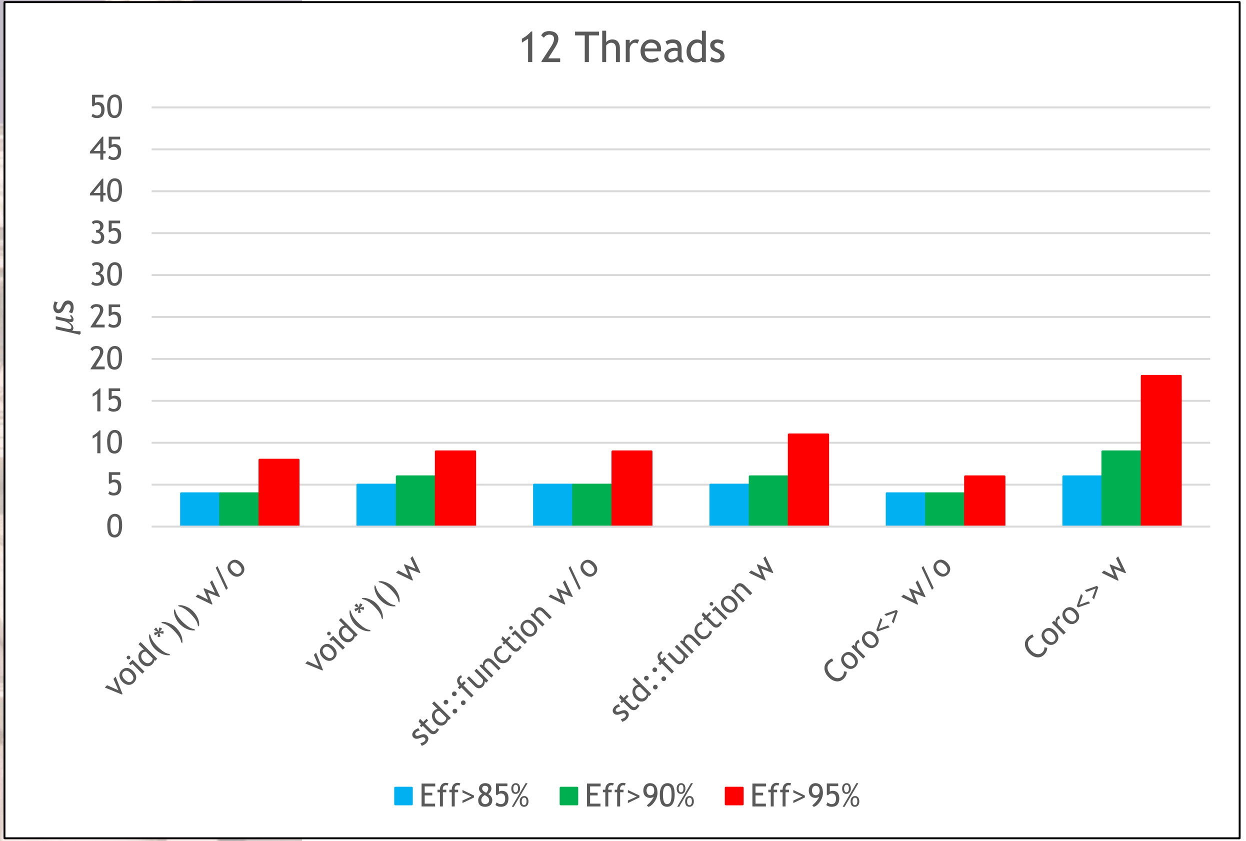
2. What is more efficient: function pointers, std::function, or coroutines?

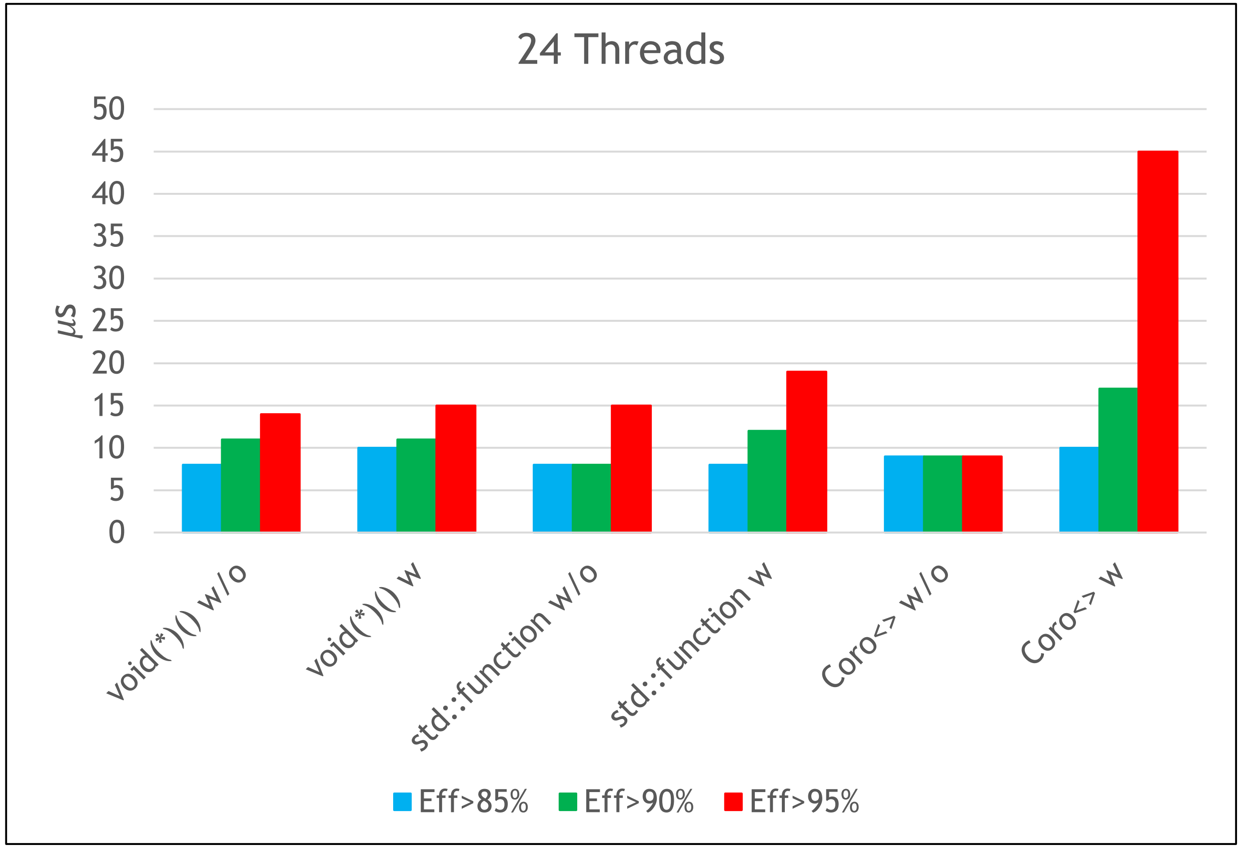
System under Test

- AMD Ryzen 7 3900X, **12/24** cores, 3793 MHz
- X570 AORUS ULTRA
- 768 KB L1, 6 MB L2, 64 MB L3 Cache
- 32 GB DDR4 RAM, 2133 MHz, DIMM
- MS Windows 10, Ver 10.0.19043

Measurement Results

Minimum Job Size (μs) to reach efficiency X %





w/o: **not** including job allocation
w: job allocation **included**

Conclusions

- Vienna Game Job System (VGJS)
- Thread Pool only, **tagged** jobs for **phases**
- Combines **coroutines** with normal **functions**
- Coroutines can return results
- Functions can interact with Coros, but complications
- Good efficiency for larger amounts of threads
- **Allocating** coroutines comes with a price

Thank you!

Any Questions?

Reach me at

helmut.hlavacs@univie.ac.at
http://entertain.univie.ac.at/~hlavacs/
https://github.com/hlavacs/ViennaGameJobSystem
https://www.linkedin.com/in/helmut-hlavacs-972b9aa/