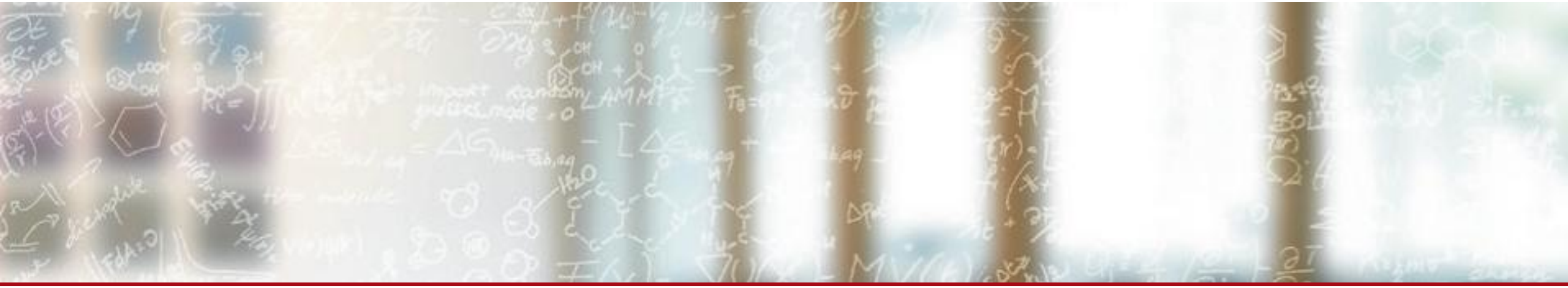
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Advanced C++ for HPC: Multithreaded task system

Nora Abi Akar, CSCS (nora.abiakar@cscs.ch)
11-13 October, 2021

Resources: Better Code: Concurrency, Sean Parent

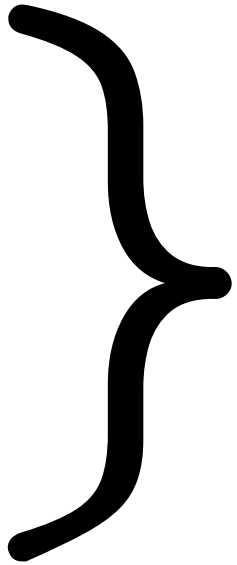
Outline

- Why use a task system?
- Design and Implementation
  - Single queue
  - Multiple queues
- Performance Comparison
- Exceptions
  - How to handle them
  - Early exits
- Questions

Why use a multi-threaded task system?

- **Application**:
  - Many **tasks**, of varying **sizes** that can execute **concurrently** and in **parallel**.

- **Requirements**:
  - Machine resources fully utilized.
  - Minimal overhead.
  - Correct handling of exceptions.
  - Early exit in case of exceptions.
  - No external libraries.
  - Short and simple code.



Task queue(s) on a thread pool

(Simple) Motivating examples

```
std::vector<int> v0(n, -1);
std::vector<std::vector<int>> v1(n, std::vector<int>(m, -1));
```

- Want to set `v0[i] = i` and `v1[i][j] = i + j`

```
task_system ts;
```

```
// Set v[i] = i
parallel_for::apply(0, n, &ts, [&v0] (int i) { v0[i] = i; });
```

```
// Set v[i][j] = i + j
parallel_for::apply(0, n, &ts, [&v1](int i) {
    auto &w = v1[i];
    parallel_for::apply(0, m, &ts, [&](int j) { w[j] = i + j; });
});
```

Existing Libraries

- Intel TBB
  - Open source; supports many platforms
- HPX
  - Open source; supports many platforms
- Parallel Patterns Library (PPL)
  - Windows
- libdispatch
  - Open source; Linux and Android

This talk can be viewed as a reference implementation

Multithreading in the standard library

Required from namespace std

- std::**vector**
- std::**deque**
- std::**function**
- std::**exception_ptr**

- std::**atomic**
- std::**mutex**
- std::**unique_lock**
- std::**thread**
- std::**condition_variable**

- std::**move**
- std::**forward**
- std::**decay**

Note: std:: has been elided in the code snippets.

cppreference.com Create account Search

Page Discussion View Edit History

C++ Thread support library std::thread

std::thread

Defined in header <thread>

```
class thread;        (since C++11)
```

The class thread represents a single thread of execution. Threads allow multiple functions to execute concurrently.

Threads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a constructor argument. The return value of the top-level function is ignored and if it terminates by throwing an exception, std::terminate is called. The top-level function may communicate its return value or an exception to the caller via std::promise or by modifying shared variables (which may require synchronization, see std::mutex and std::atomic)

std::condition_variable

Defined in header <condition_variable>

```
class condition_variable;      (since C++11)
```

The condition_variable class is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until another thread both modifies a shared variable (the *condition*), and notifies the condition_variable.

The thread that intends to modify the variable has to

1. acquire a std::mutex (typically via std::lock_guard)
2. perform the modification while the lock is held
3. execute notify_one or notify_all on the std::condition_variable (the lock does not need to be held for notification)

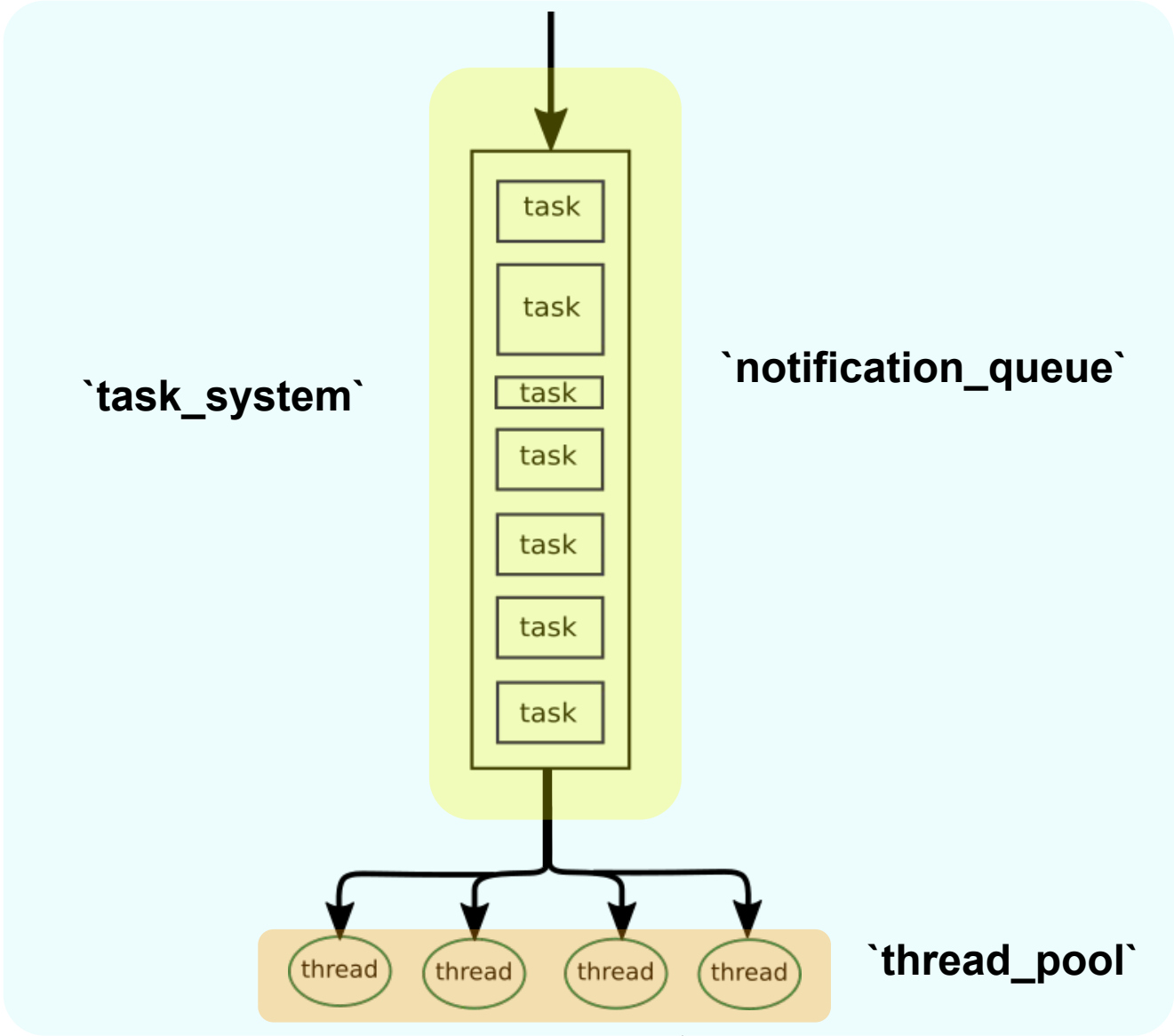
Even if the shared variable is atomic, it must be modified under the mutex in order to correctly publish the modification to the waiting thread.

Any thread that intends to wait on std::condition_variable has to

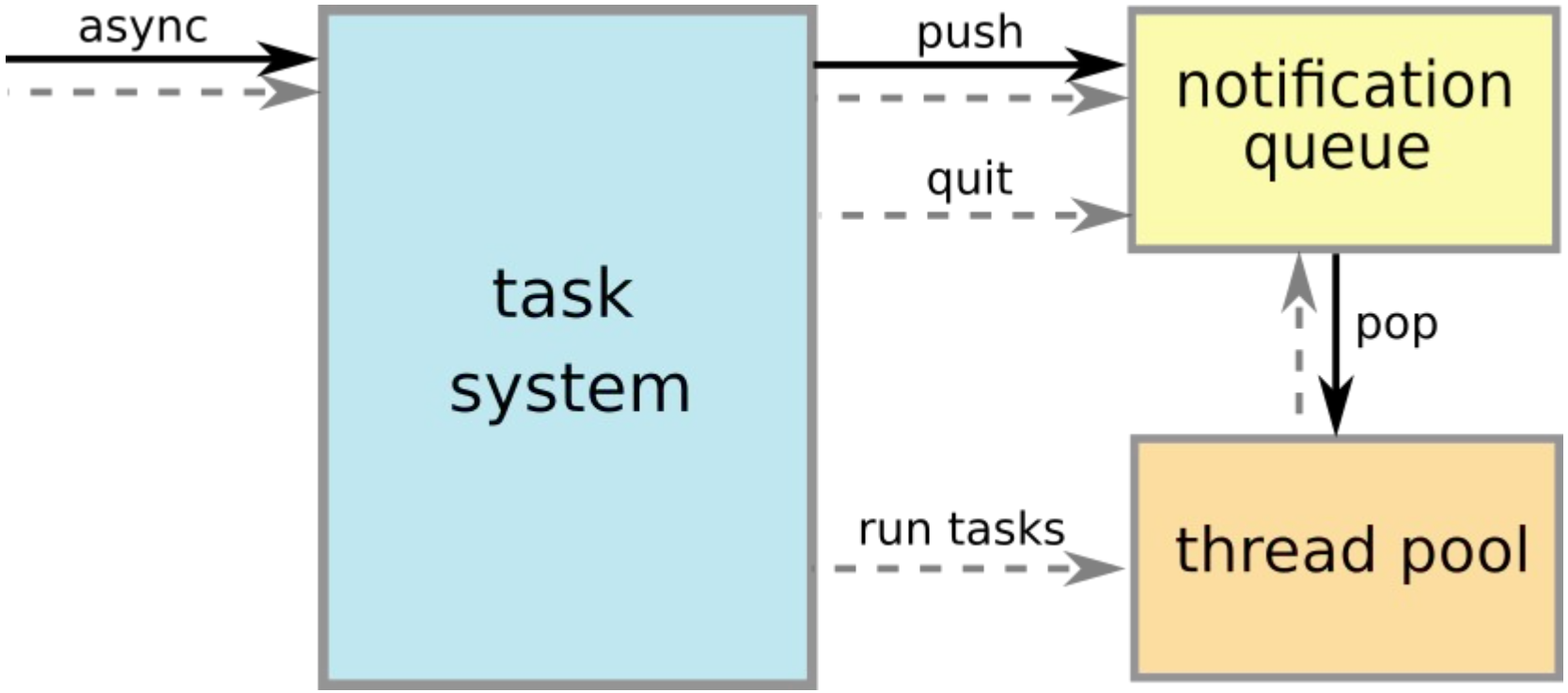
1. acquire a std::unique_lock<std::mutex>, on the same mutex as used to protect the shared variable
2. execute wait, wait_for, or wait_until. The wait operations atomically release the mutex and suspend the execution of the thread.
3. When the condition variable is notified, a timeout expires, or a spurious wakeup occurs, the thread is awakened, and the mutex is atomically reacquired. The thread should then check the condition and resume waiting if the wake up was spurious.

Design and Implementation: Single Task queue

Design overview: Single Task Queue



Design overview: Single Task Queue



Blocking Concurrent Queue

Vector of std::thread

Single Queue System: `task_system`

```
using lock = unique_lock<mutex>;
using task = function<void()>;


class task_system {
    vector<thread>              threads_;
    notification_queue          q_;

public:
    task_system(int nthreads) {...}

    ~task_system() {...}

    void run_tasks_loop() {...}

    void async(task tsk) {...}
};
```

Single Queue System: `task_system`

```
using lock = unique_lock<mutex>;
using task = function<void()>;
```

Single Queue System: `task_system`

```
using lock = unique_lock<mutex>;
using task = function<void()>;

class task_system {
    vector<thread>              threads_;
    notification_queue          q_;
```

Single Queue System: `task_system`

```
using lock = unique_lock<mutex>;
using task = function<void()>;

class task_system {
    vector<thread>              threads_;
    notification_queue          q_;

public:
    task_system(int nthreads) {
        for (unsigned i = 1; i < nthreads; i++) {
            threads_.emplace_back( [this]  {  run_tasks_loop();  } );
        }
    }
```

Single Queue System: `task_system`

```
using lock = unique_lock<mutex>;
using task = function<void()>;

class task_system {
    vector<thread>              threads_;
    notification_queue          q_;

public:
    task_system(int nthreads) {...}

    ~task_system() {
        q_.quit();
        for (auto& e: threads_) e.join();
    }
```

Single Queue System: `task_system`

```
using lock = unique_lock<mutex>;
using task = function<void()>;

class task_system {
    vector<thread>              threads_;
    notification_queue          q_;

public:
    task_system(int nthreads) {...}

    ~task_system() {...}

    void run_tasks_loop() {
        while (true) {
            task tsk = q_.pop();
            if (!tsk) break;
             tsk();
        }
    }
```

Single Queue System: `task_system`

```
using lock = unique_lock<mutex>;
using task = function<void()>;

class task_system {
    vector<thread>              threads_;
    notification_queue          q_;

public:
    task_system(int nthreads) {...}

    ~task_system() {...}

    void run_tasks_loop() {...}

    void async(task tsk) {
        q_.push(std::move(tsk));
    }
};
```

Single Queue System: `notification_queue`

```
using lock = unique_lock<mutex>;
using task = function<void()>;

class notification_queue {
    deque<task>          q_tasks_;
    mutex                q_mutex_;
    condition_variable   q_tasks_available_;
    bool                 quit_ = false;

public:
    task pop() {...}

    void push(task&& tsk) {...}

    void quit() {...}

}
```

Single Queue System: `notification_queue`

```
using lock = unique_lock<mutex>;
using task = function<void()>;
```

Single Queue System: `notification_queue`

```
using lock = unique_lock<mutex>;
using task = function<void()>;

class notification_queue {
    deque<task>          q_tasks_;
    mutex                q_mutex_;
    condition_variable   q_tasks_available_;
    bool                 quit_ = false;
```

Single Queue System: `notification_queue`

```
using lock = unique_lock<mutex>;
using task = function<void()>;

class notification_queue {
    deque<task>          q_tasks_;
    mutex                q_mutex_;
    condition_variable   q_tasks_available_;
    bool                 quit_ = false;

public:
    task pop() {
        task tsk;
        lock q_lock{q_mutex_};
        while (q_tasks_.empty() && !quit_) {
            q_tasks_available_.wait(q_lock);
        }
        if (!q_tasks_.empty()) {
            tsk = std::move(q_tasks_.front());
            q_tasks_.pop_front();
        }
        return tsk;
    }
```

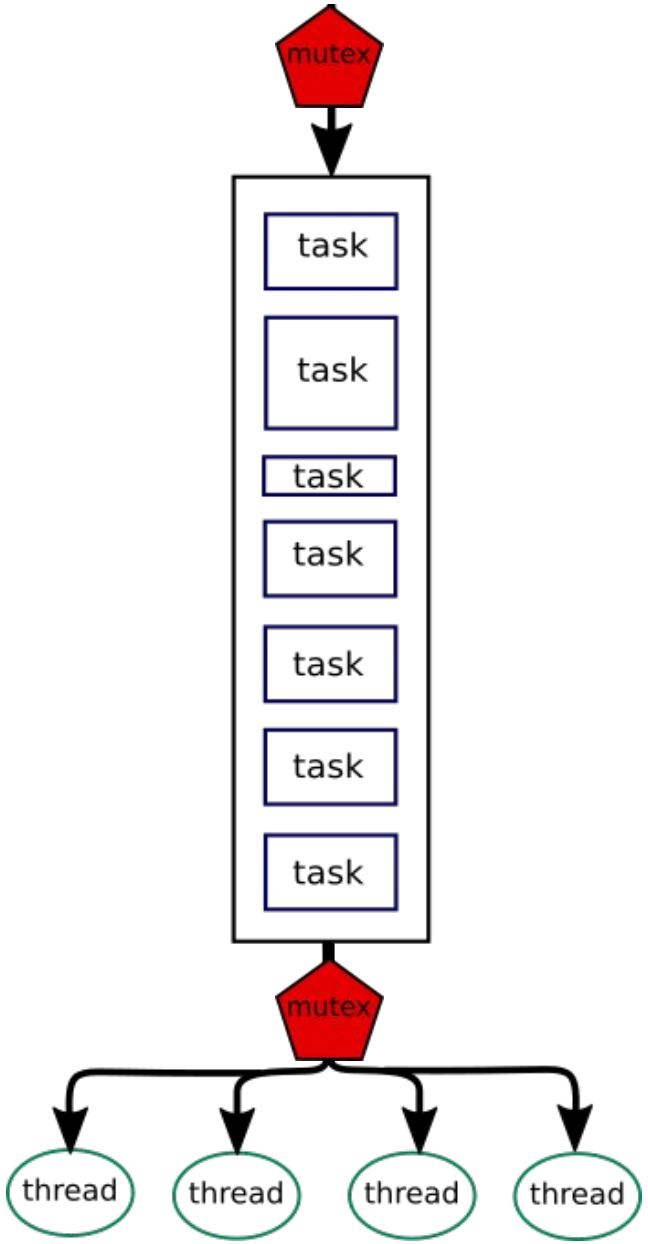
Single Queue System: `notification_queue`

```
using lock = unique_lock<mutex>;
using task = function<void()>;

class notification_queue {
    deque<task>          q_tasks_;
    mutex                q_mutex_;
    condition_variable   q_tasks_available_;
    bool                 quit_ = false;

public:
    task pop() {...}

    void push(task&& tsk) {
        {
            lock q_lock{q_mutex_};
            q_tasks_.push_back(std::move(tsk));
        }
        q_tasks_available_.notify_all();
    }
```

Single Queue System: `notification_queue`

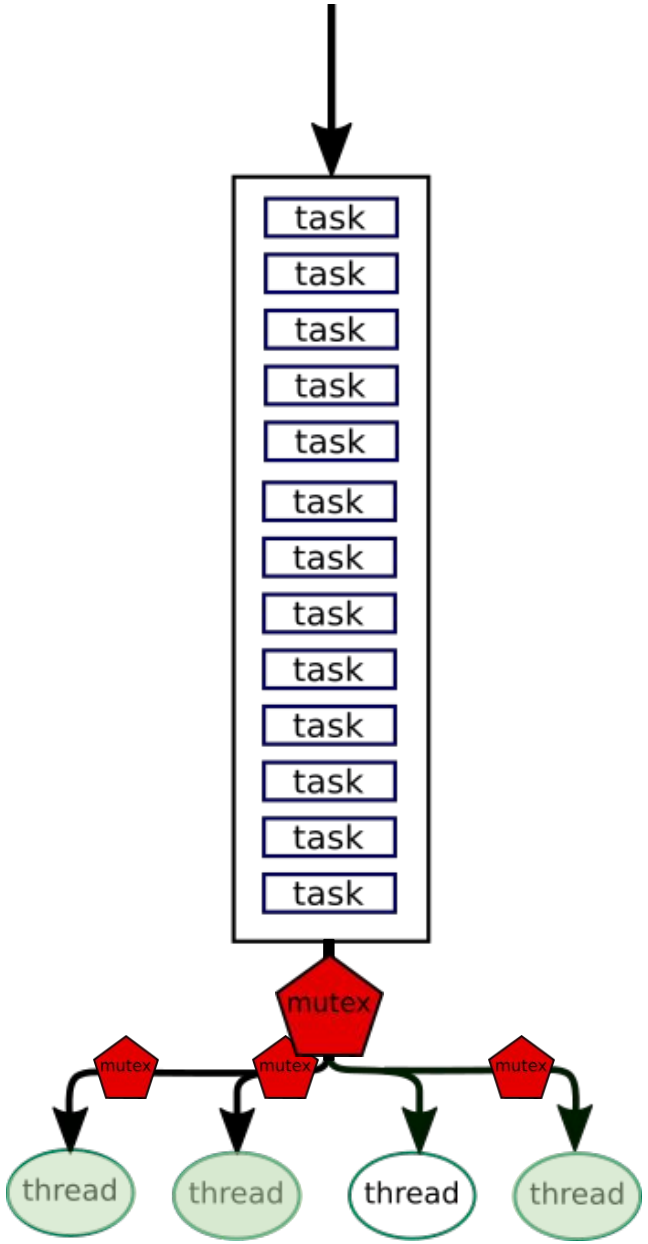
```
using lock = unique_lock<mutex>;
using task = function<void()>;

class notification_queue {
    deque<task>          q_tasks_;
    mutex                q_mutex_;
    condition_variable   q_tasks_available_;
    bool                 quit_ = false;

public:
    task pop() {...}

    void notification_queue::push(task&& tsk) {...}

    void quit() {
        {
            lock q_lock{q_mutex_};
            quit_ = true;
        }
        q_tasks_available_.notify_all();
    }
```

Single Queue System



Mutex synchronization adds a serialization overhead

Single Queue System

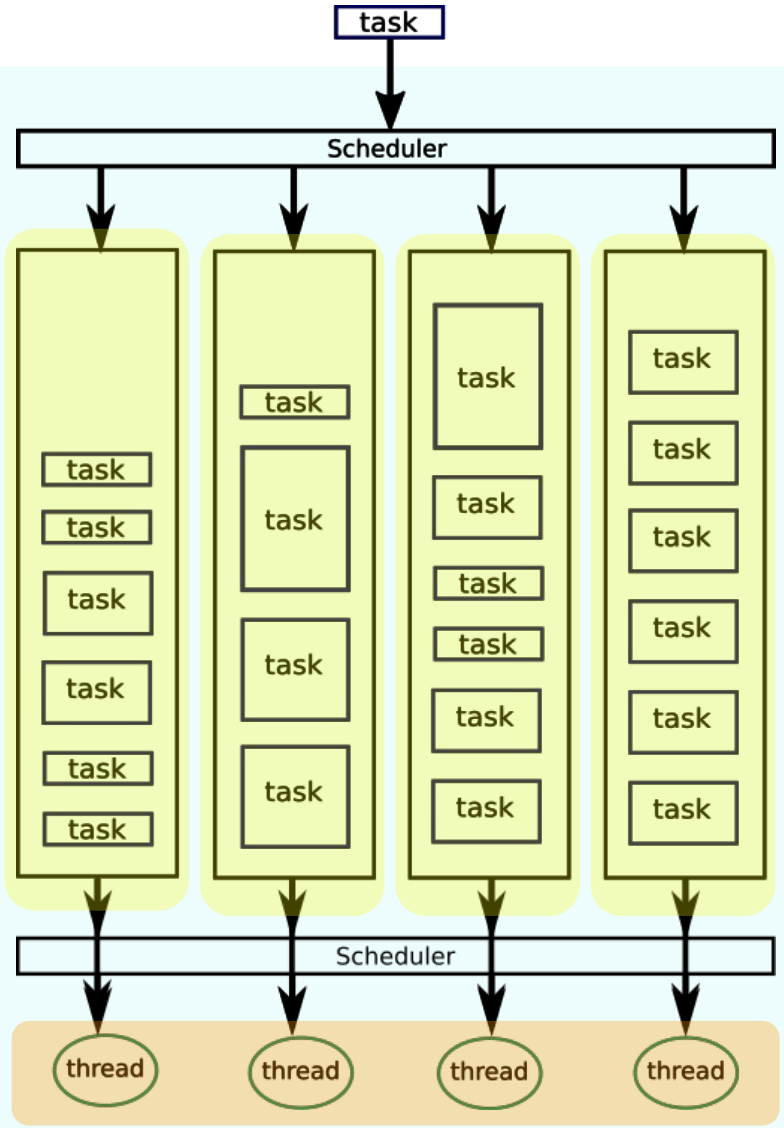


Mutex synchronization adds a serialization overhead

Design and Implementation: Multiple Task queues

Multiple Task Queues

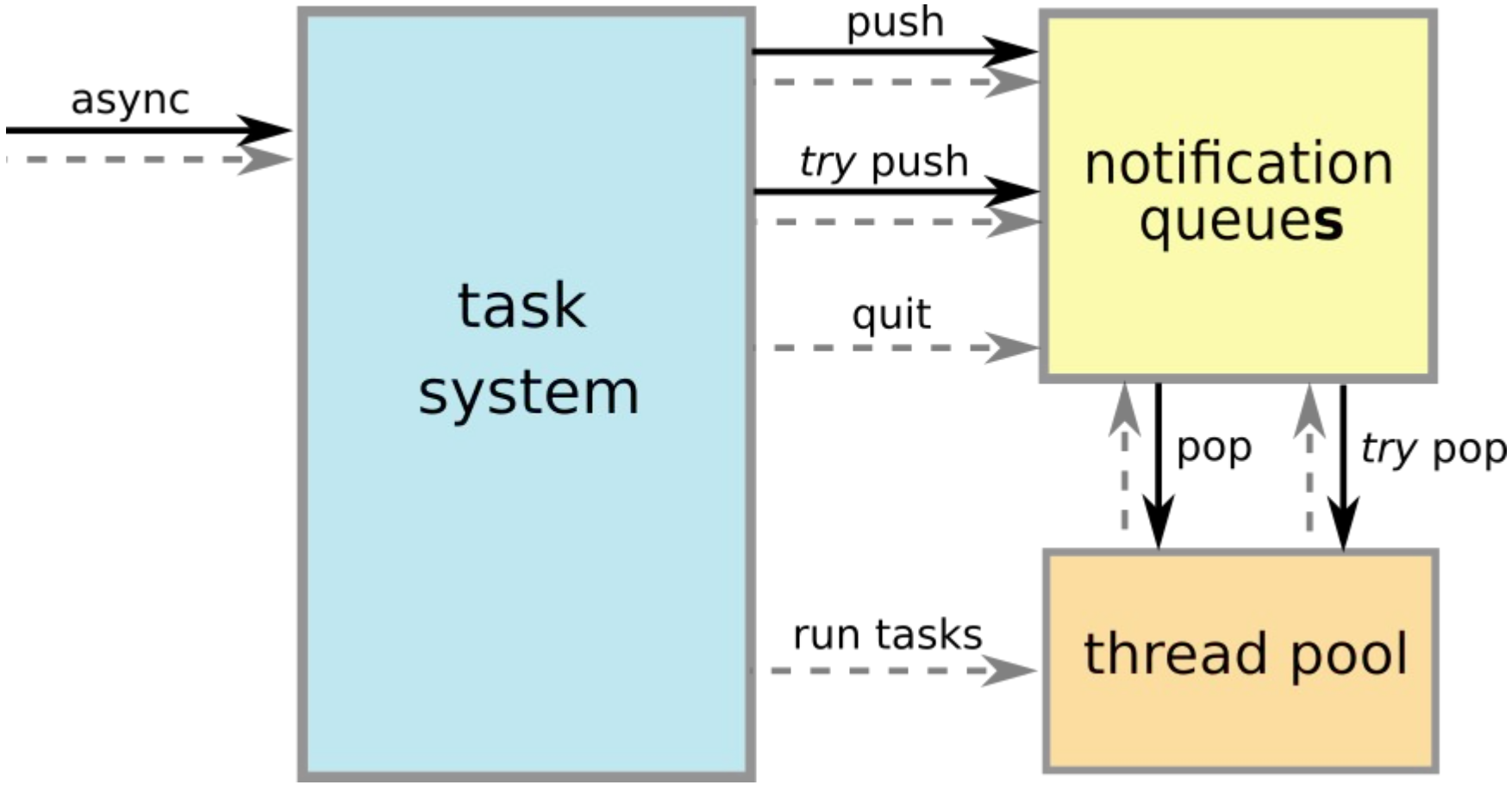
`task_system`



`notification_queue`

`thread_pool`

Multiple Task Queues



Multiple Queue System: `notification_queue`

```
using lock = unique_lock<mutex>;
using task = function<void()>;

class notification_queue {
    deque<task>          q_tasks_;
    mutex                q_mutex_;
    condition_variable   q_tasks_available_;
    bool                 quit_ = false;

public:
    task pop() {...}

    void push(task&& tsk) {...}

    void quit() {...}

    task try_pop() {...}

    bool try_push(task&& tsk) {...}
```

Multiple Queue System: `notification_queue`

```
using lock = unique_lock<mutex>;
using task = function<void()>;

class notification_queue {
    deque<task>          q_tasks_;
    mutex                q_mutex_;
    condition_variable   q_tasks_available_;
    bool                 quit_ = false;

public:
    task try_pop() {
        task tsk;
        lock q_lock{q_mutex_, std::try_to_lock};
        if (q_lock && !q_tasks_.empty()) {
            tsk = std::move(q_tasks_.front());
            q_tasks_.pop_front();
        }
        return tsk;
    }
```

Multiple Queue System: `notification_queue`

```
using lock = unique_lock<mutex>;
using task = function<void()>;

class notification_queue {
    deque<task>          q_tasks_;
    mutex                q_mutex_;
    condition_variable   q_tasks_available_;
    bool                 quit_ = false;

public:
    task try_pop() {...}

    bool try_push(task&& tsk) {
        {
            lock q_lock{q_mutex_, std::try_to_lock};
            if (!q_lock) return false;
            q_tasks_.push_back(std::move(tsk));
        }
        q_tasks_available_.notify_all();
        return true;
    }
```

Multiple Queue System: `task_system`

```
class task_system {
    vector<thread>                  threads_;
    vector<notification_queue>      q_;
    unsigned                        count_;       // total number of threads
    atomic<unsigned>                index_{0};    // total number of tasks pushed in all queues

public:
    task_system(int nthreads) {...}

    ~task_system() {...}

    void run_tasks_loop(int idx) {...}

    void async(task tsk) {...}

    void try_run_task() {...}

};
```

Multiple Queue System: `task_system`

```
class task_system {
    vector<thread>                  threads_;
    vector<notification_queue>      q_;
    unsigned                        count_;       // total number of threads
    atomic<unsigned>                index_{0};    // total number of tasks pushed in all queues
```

Multiple Queue System: `task_system`

```
class task_system {
    vector<thread>                  threads_;
    vector<notification_queue>      q_;
    unsigned                        count_;       // total number of threads
    atomic<unsigned>                index_{0};    // total number of tasks pushed in all queues

public:
    task_system(int nthreads) : count_(nthreads), q_(nthreads) {
        for (unsigned i = 1; i < count_; i++) {
            threads_.emplace_back( [this, i] { run_tasks_loop(i); } );
        }
    }
```

Multiple Queue System: `task_system`

```
class task_system {
    vector<thread>                  threads_;
    vector<notification_queue>      q_;
    unsigned                        count_;      // total number of threads
    atomic<unsigned>                index_{0};   // total number of tasks pushed in all queues

public:
    task_system(int nthreads) {...}

    ~task_system() {
        for (auto& e: q_) e.quit();
        for (auto& e: threads_) e.join();
    }
```

Multiple Queue System: `task_system`

```
class task_system {
    vector<thread>                  threads_;
    vector<notification_queue>      q_;
    unsigned                        count_;      // total number of threads
    atomic<unsigned>                index_{0};   // total number of tasks pushed in all queues

public:
    task_system(int nthreads) {...}

    ~task_system() {...}

    void run_tasks_loop(int idx) {
        while (true) {
            task tsk;
            for (unsigned n = 0; n != count_; n++) {
                tsk = q_[(idx + n) % count_].try_pop();
                if (tsk) break;
            }
            if (!tsk) tsk = q_[idx].pop();
            if (!tsk) break;
            tsk();
        }
    }
```

Multiple Queue System: `task_system`

```
class task_system {
    vector<thread>                  threads_;
    vector<notification_queue>      q_;
    unsigned                        count_;      // total number of threads
    atomic<unsigned>                index_{0};   // total number of tasks pushed in all queues

public:
    task_system(int nthreads) {...}

    ~task_system() {...}

    void run_tasks_loop(int idx) {...}

    void async(task tsk) {
        auto i = index_++;
        for (unsigned n = 0; n != count_; n++) {
            if (q_[(i + n) % count_].try_push(tsk)) return;
        }
        q_[i % count_].push(std::move(tsk));
    }
```

Multiple Queue System: `task_system`

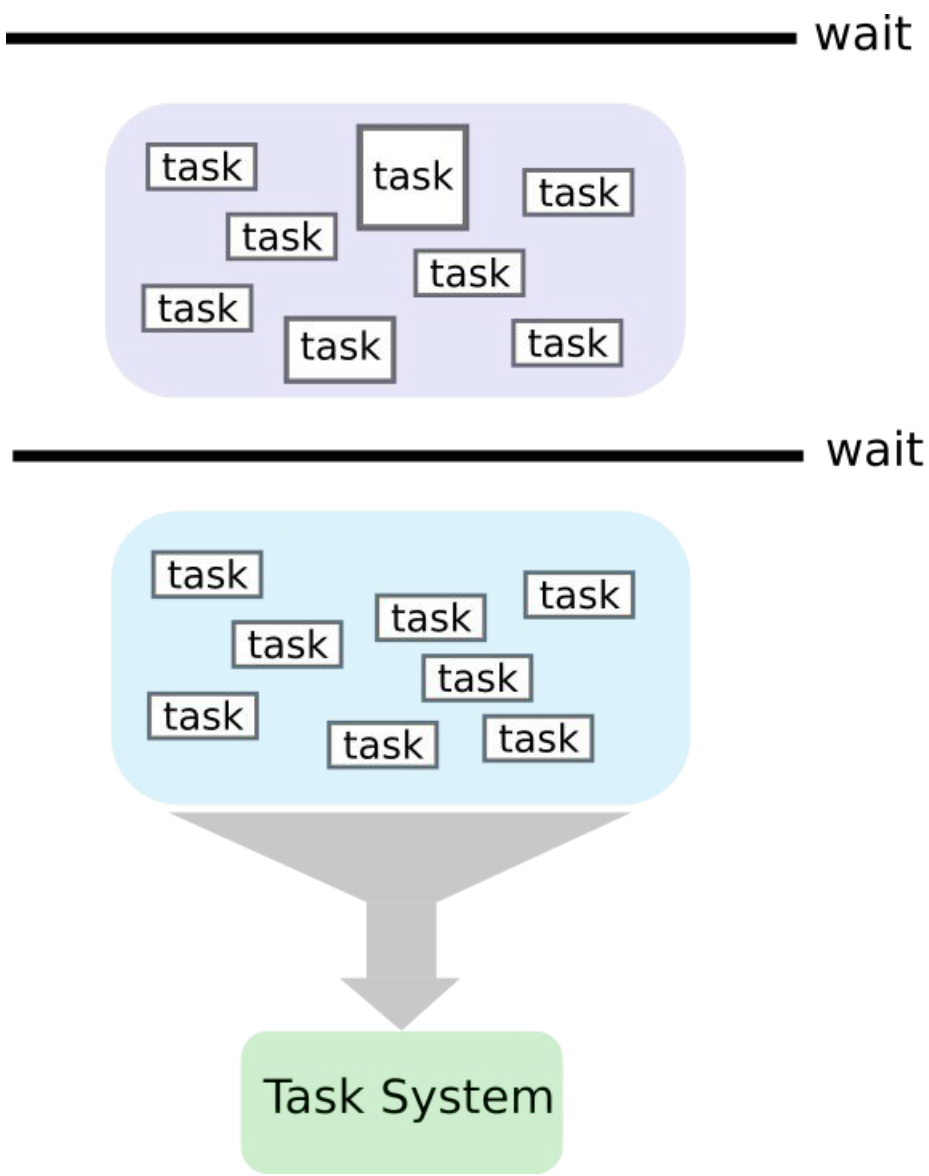
```
class task_system {
    vector<thread>                  threads_;
    vector<notification_queue>      q_;
    unsigned                        count_;      // total number of threads
    atomic<unsigned>                index_{0};   // total number of tasks pushed in all queues

public:
    task_system(int nthreads) {...}

    ~task_system() {...}

    void run_tasks_loop(int idx) {...}

    void async(task tsk) {...}
```

```
    void try_run_task() {
        task tsk;
        for (unsigned n = 0; n != count_; n++) {
            tsk = q_[n].try_pop();
            if (tsk) {
                tsk();
                break;
            }
        }
    }
```

Design and Implementation: Task groups

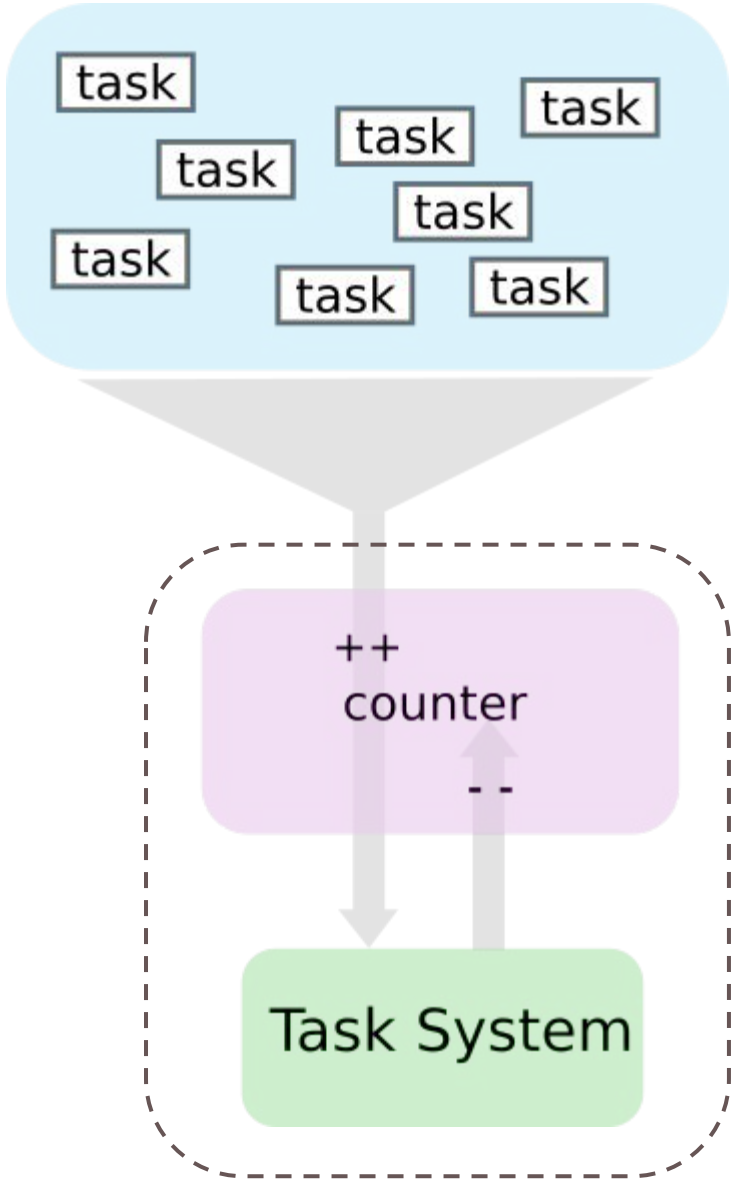
`task_group`

- Frequently, groups of tasks need to be serialized
- We need a way to `wait` for a signal from the task system that a group of tasks has been executed successfully



`task_group`

- One more layer of abstraction **`task_group`**
- Represents a group of tasks that we can *wait* on.
- Keep track number of tasks “in flight” in the system, when that number is zero, all tasks have been executed.

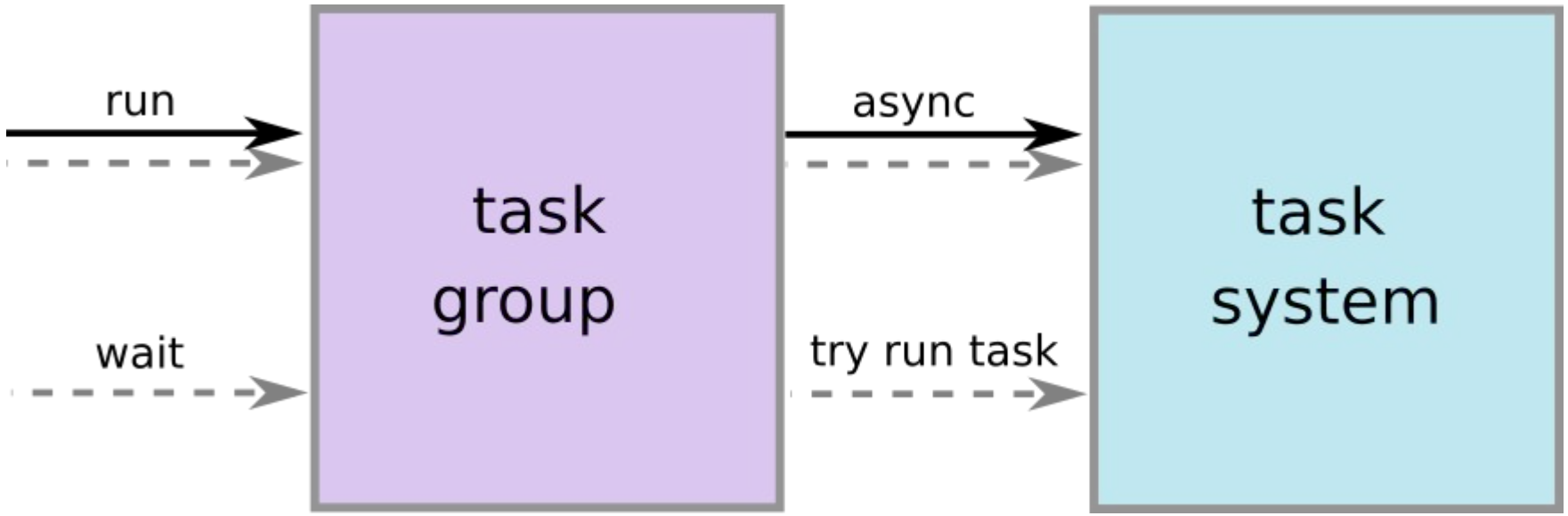


`run`: Push tasks in task system and start execution

`task_group`

`wait`: wait until task counter==0; help with executing tasks while waiting

`task_group`



`task_group`

```
class task_group {
    atomic<size_t>          in_flight_{0};
    task_system*            task_system_;

public:
    task_group(task_system* ts);

    ~task_group() ;

    void run(F&& f);

    void wait();
};
```

`task_group`

```
class task_group {
    atomic<size_t>          in_flight_{0};
    task_system*            task_system_;
```

`task_group`

```
class task_group {
    atomic<size_t>          in_flight_{0};
    task_system*            task_system_;

public:
    task_group(task_system* ts):
        task_system_{ts}
    {}
```

`task_group`

```
class task_group {
    atomic<size_t>          in_flight_{0};
    task_system*            task_system_;

public:
    task_group(task_system* ts) {...}

    ~task_group() {
        wait();
    }
```

`task_group`

```
class task_group {
    atomic<size_t>          in_flight_{0};
    task_system*            task_system_;

public:
    task_group(task_system* ts) {...}

    ~task_group() {...}

    void run(F&& f);

    void wait() {
        while (in_flight_) {
            task_system_->try_run_task();
        }
    }
};
```

`task_group`

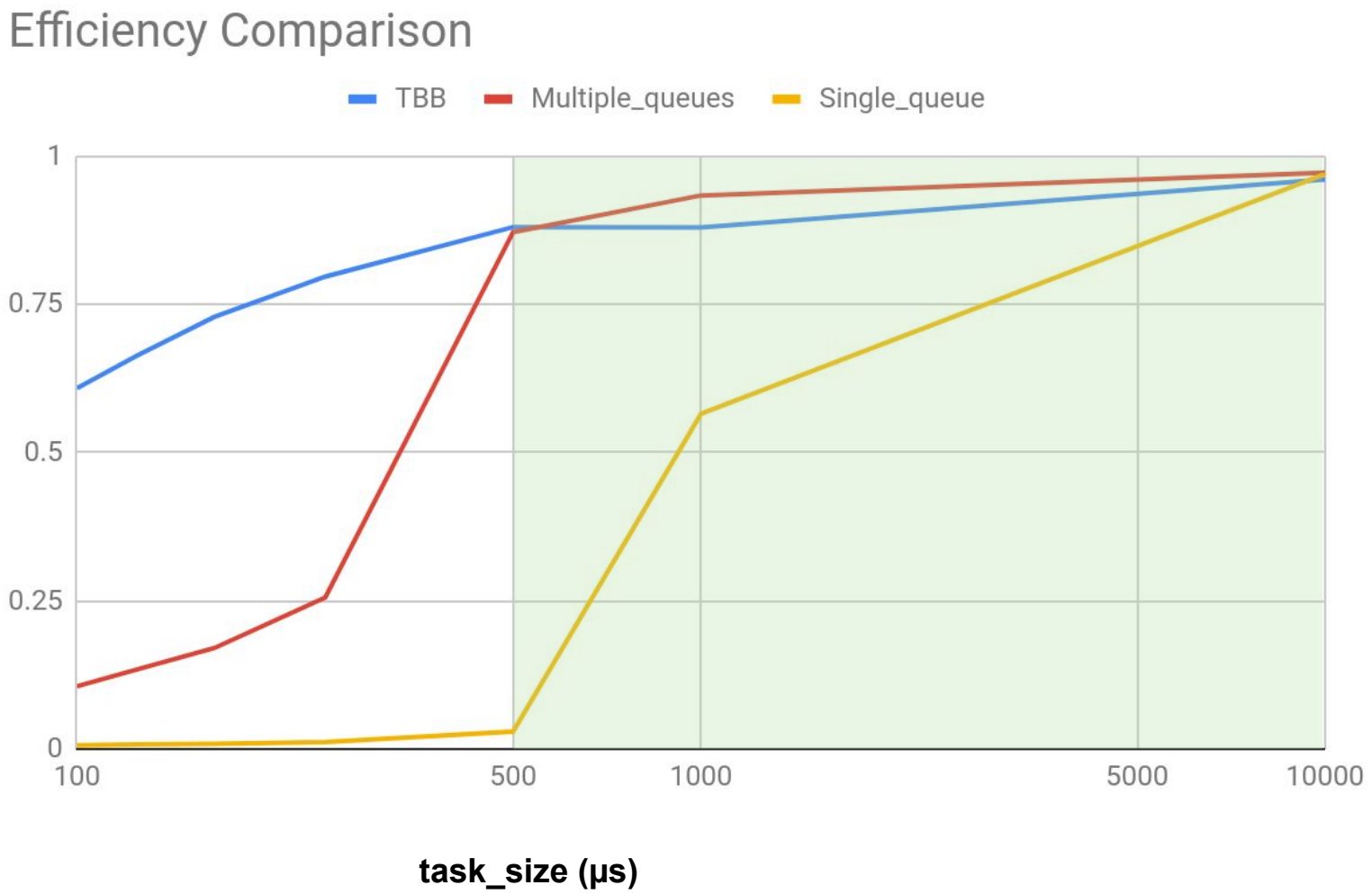
```
class task_group {
    atomic<size_t>          in_flight_{0};
    task_system*            task_system_;

public:
    task_group(task_system* ts) {...}

    ~task_group() {...}

    template<typename F>
    void run(F&& f) {
        ++in_flight_;
        task_system_->async(make_wrapped_function(std::forward<F>(f), in_flight_));
    }

    void wait() {...}
};
```

`task_group`

```
template <typename F>
class wrap {
      F f_;
      atomic<std::size_t>& counter_;

public:
     template <typename F2>
     explicit wrap(F2&& other, atomic<size_t>& c):
           f_(std::forward<F2>(other)),
           counter_(c)  {}
     // copy and move constructors

     void operator()() {
       f_();
       --counter_;
     }
};

template <typename F>
using callable = typename std::decay<F>::type;

template <typename F>
wrap<callable<F>> make_wrapped_function(F&& f, atomic<size_t>& c) {
  return wrap<callable<F>>(std::forward<F>(f), c);
}
```

Performance

Efficiency = (task_size * num_tasks)/ execution_time



Exceptions

Why use a multithreaded task system?

- **Application**:
  - Many **tasks**, of varying **sizes** that can execute **concurrently** and in **parallel**.

- **Requirements**:
  - ✓ Machine resources fully utilized.
  - ✓ Minimal overhead.
  - ? *Correct handling of exceptions*.
  - ? *Early exit in case of exceptions*.
  - ✓ No external libraries.
  - ✓ Short and simple code.

Exception Handling

- One of the tasks in the queue can raise an **exception**
- We want to stop execution and return the exception
  - There's no point in executing the additional tasks
- If multiple tasks raise exceptions, return any of the exceptions
- Stopping the execution doesn't have to happen immediately after the exception is raised
  - We have some leeway
  - Can execute a few extra tasks
  - This relaxes our synchronization constraints
- Exception handling done separately from the task system in **`task_group`**

Exception Handling: `task_group`

```
class task_group {
    atomic<size_t>              in_flight_{0};
    task_system*                task_system_;
    exception_state             exception_status_;

public:
    task_group(task_system* ts);

    ~task_group() ;

     void run(F&& f);

     void wait();
    };
```

Exception Handling: `task_group`

```
class task_group {
    atomic<size_t>              in_flight_{0};
    task_system*                task_system_;
    exception_state             exception_status_;

public:
    task_group(task_system* ts) {...}

    ~task_group() {...}

    template<typename F>
    void run(F&& f) {...}

    void wait() {
        while (in_flight_) {
            task_system_->try_run_task();
        }

        if (auto ex = exception_status_.reset()) {
            std::rethrow_exception(ex);
        }
    }

};
```

Exception Handling: `task_group`

```
class task_group {
    atomic<size_t>              in_flight_{0};
    task_system*                task_system_;
    exception_state             exception_status_;

public:
    task_group(task_system* ts) {...}

    ~task_group() {...}

    template<typename F>
    void run(F&& f) {
        ++in_flight_;
        task_system_->async(make_wrapped_function(std::forward<F>(f), in_flight_, exception_status_));
    }

    void wait() {...}
};
```

Exception Handling: `wrap`

```
template <typename F>
class wrap {
    F                       f_;
    atomic<size_t>&         counter_;
    exception_state&        exception_status_;

public:
    template <typename F2>
    explicit wrap(F2&& other, atomic<std::size_t>& c, exception_state& ex):
        f_(std::forward<F2>(other)), counter_(c),  exception_status_(ex) {}
    // copy and move constructors

    void operator()() {...}
};

template <typename F>
using callable = typename std::decay<F>::type;

template <typename F>
wrap<callable<F>> make_wrapped_function(F&& f, atomic<std::size_t>& c, exception_state& ex) {
  return wrap<callable<F>>(std::forward<F>(f), c, ex);
```

Exception Handling: `wrap`

```
template <typename F>
class wrap {
    F                       f_;
    atomic<size_t>&         counter_;
    exception_state&        exception_status_;

public:
    template <typename F2>
    explicit wrap(F2&& other, atomic<std::size_t>& c, exception_state& ex):
        f_(std::forward<F2>(other)), counter_(c),  exception_status_(ex) {}
    // copy and move constructors

    void operator()() {
        if (!exception_status_) {
            try {
                f_();
            }
            catch (...) {
                exception_status_.set(std::current_exception());
            }
        }
        --counter_;
    }
};
```

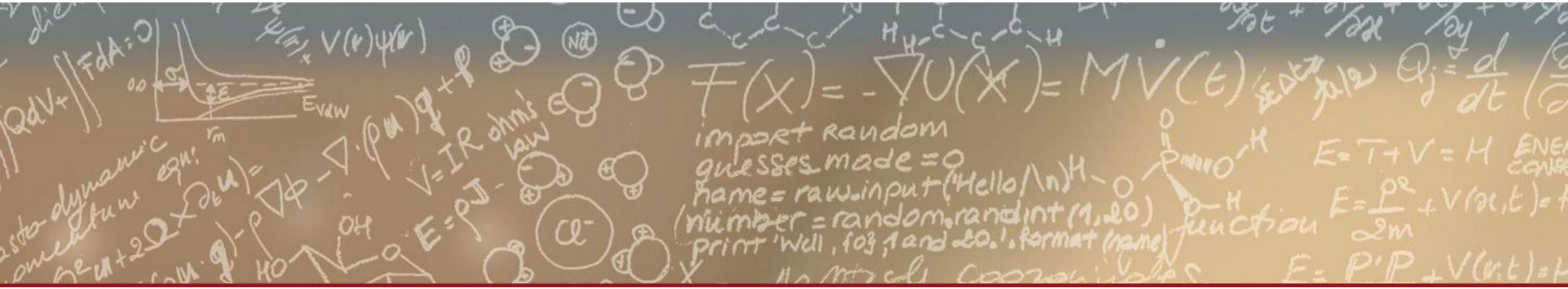
Exception Handling: `exception_state`

```
struct exception_state {
    atomic<bool>      error_{false};
    exception_ptr     exception_;
    mutex             mutex_;

    operator bool() const {
        return error_.load(std::memory_order_relaxed);
    }

    void set(exception_ptr ex) {
        error_.store(true, std::memory_order_relaxed);
        lock ex_lock{mutex_};
        exception_ = std::move(ex);
    }

    exception_ptr reset() {
        auto ex = std::move(exception_);
        error_.store(false, std::memory_order_relaxed);
        exception_ = nullptr;
        return ex;
    }
};
```

Why use a multithreaded task system?

- **Application**:
  - Many **tasks**, of varying **sizes** that can execute **concurrently** and in **parallel**.

- **Requirements**:
  - ✓ Machine resources fully utilized.
  - ✓ Minimal overhead.
  - ✓ Correct handling of exceptions.
  - ✓ Early exit in case of exceptions.
  - ✓ No external libraries.
  - ✓ Short and simple code.

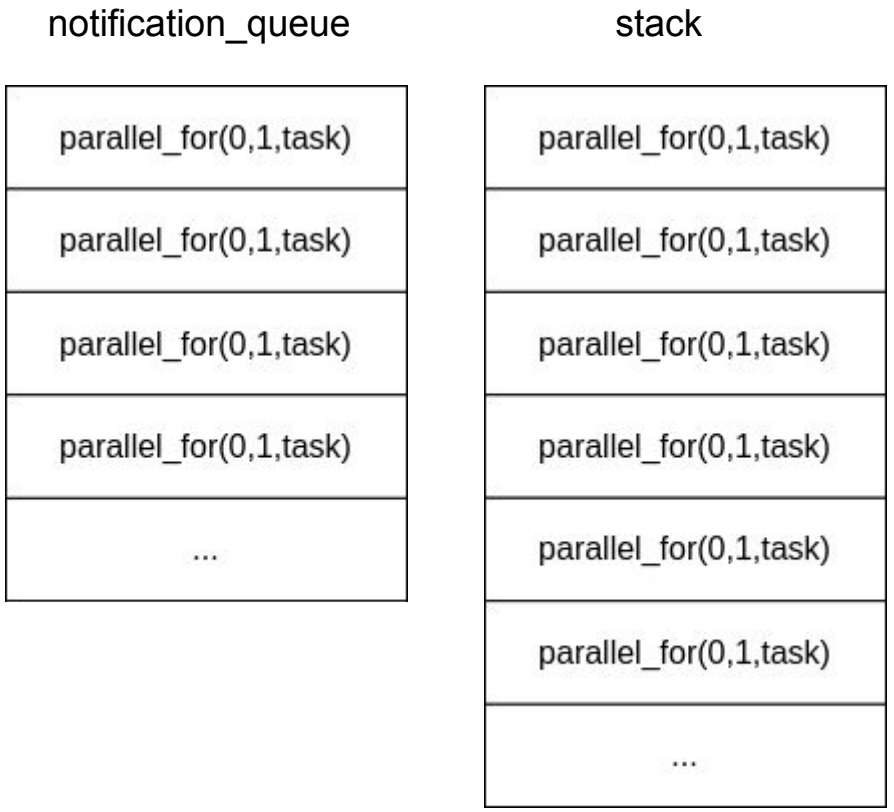


Thank you for your attention.

Parallel for loop

```
struct parallel_for {
        template <typename F>
        static void apply(int left, int right, task_system* ts, F f) {
                task_group g(ts);
                for (int i = left; i < right; ++i) {
                        g.run([=] {f(i);});
                }
                g.wait();
        }
};
```

```
parallel_for::apply(0, 20000, &ts, [&](int i) {
    parallel_for::apply(0, 1, &ts, [&](int j) { task(); });
});
```



Full implementation in C++ (including priority queues)

Arbor:

https://github.com/arbor-sim/arbor

Threading library:

https://github.com/arbor-sim/arbor/blob/master/arbor/threading/threading.cpp

https://github.com/arbor-sim/arbor/blob/master/arbor/threading/threading.hpp

Stack overflow bug and solution:

https://github.com/arbor-sim/arbor/pull/1583