ThreadPool

Generated by Doxygen 1.9.4

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 ThreadPool Class Reference

Manages a pool of worker threads to execute tasks concurrently.

`#include <ThreadPool.hpp>`

Public Member Functions

- ThreadPool ()=delete

  Deleted default constructor.
- ThreadPool (unsigned int num_threads)

  Constructs a ThreadPool with a fixed number of worker threads.
- template<class Func , class... Args>
  void submit (Func &&f, Args &&... args)

  Submits a task to the thread pool.
- void finish (bool secure=true)

  Stops the thread pool.
- bool status ()

  Returns the current state of the thread pool.
- unsigned int length ()

  Returns the number of pending tasks in the queue.
- void wait ()

  Waits until the task queue becomes empty.
- ~ThreadPool ()

  Destructor.

Data Fields

- atomic< bool > error = false

  Indicates whether any worker thread has thrown an exception.

Private Attributes

- thread * th
- mutex mtx
- queue< function< void()> > q
- unsigned int num_threads
- atomic< bool > exit_flag
- atomic< bool > * process_flag
- atomic< bool > state
- atomic< bool > waiting
- condition_variable cv
- condition_variable cv2

3.1.1 Detailed Description

Manages a pool of worker threads to execute tasks concurrently.

The ThreadPool maintains a queue of tasks. Worker threads continuously fetch and execute tasks from this queue until the pool is stopped.

Note

Tasks have no return value.

Definition at line 34 of file ThreadPool.hpp.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 ThreadPool() [1/2]

```
ThreadPool::ThreadPool ( )  [delete]
```

Deleted default constructor.

Forces the user to specify the number of threads.

3.1.2.2 ThreadPool() [2/2]

```
ThreadPool::ThreadPool (
            unsigned int num_threads )  [explicit]
```

Constructs a ThreadPool with a fixed number of worker threads.

Constructs the thread pool and starts the worker threads.

Parameters

num_threads	Number of worker threads to create.

Each worker thread runs a loop that retrieves tasks from the queue and executes them until termination is requested.

Parameters

num_threads	Number of worker threads to create.

Worker thread routine.

Each worker thread continuously waits for tasks in the queue and executes them while the pool remains active and no exit has been requested.

Parameters

process_number	Index of the worker thread.

Definition at line 17 of file ThreadPool.cpp.

```
00018     :  num_threads(num_threads) {
00019
00020 /**
00021 * @brief Worker thread routine.
00022 *
00023 * Each worker thread continuously waits for tasks
00024 * in the queue and executes them while the pool
00025 * remains active and no exit has been requested.
00026 *
00027 * @param process_number Index of the worker thread.
00028 */
00029     function<void(unsigned int)> f = [&](unsigned int process_number) {
00030
00031         function<void()> task;
00032
00033         unique_lock<mutex> lock(this->mtx);
00034
00035         while (!this->exit_flag.load() && this->state.load()) {
00036
00037             cv.wait(lock, [&]() {
00038                 return !this->q.empty()
00039                        || this->exit_flag.load()
00040                        || !this->state.load();
00041             });
00042
00043             if (this->exit_flag.load() || !this->state.load()) {
00044                 lock.unlock();
00045                 break;
00046             }
00047
00048             task = move(this->q.front());
00049             this->q.pop();
00050
00051             lock.unlock();
00052
00053             this->process_flag[process_number] = true;
00054             try {
00055                 task();
00056             } catch (...)  {
00057                 this->error = true;
00058             }
00059             this->process_flag[process_number] = false;
00060
00061             lock.lock();
00062             if (this->waiting.load()) {
00063                 this->cv2.notify_one();
00064             }
00065         }
00066
00067         if (lock.owns_lock()) {
00068             lock.unlock();
00069         }
```

```
00070     };
00071
00072     this->exit_flag = false;
00073     this->waiting = false;
00074
00075     this->th = new (nothrow) thread[num_threads];
00076     this->process_flag = new (nothrow) atomic<bool>[num_threads];
00077
00078     if (this->th != nullptr && this->process_flag != nullptr) {
00079         this->state = true;
00080
00081         try {
00082             for (unsigned int i = 0; i < this->num_threads; i++) {
00083                 this->th[i] = thread(f, i);
00084                 this->process_flag[i] = false;
00085             }
00086         } catch (exception&) {
00087             this->state = false;
00088             // Failed to start one or more worker threads
00089         }
00090     } else {
00091         this->state = false;
00092     }
00093 }
```

3.1.2.3 ~ThreadPool()

```
ThreadPool::~ThreadPool ( )
```

Destructor.

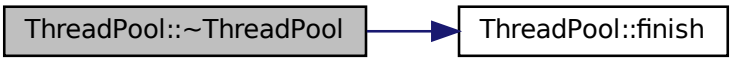
Stops the thread pool and releases all allocated resources.

Ensures that the thread pool is stopped and all allocated resources are released.

Definition at line 149 of file ThreadPool.cpp.

```
00149                         {
00150     this->finish();
00151     try {
00152         delete[] this->th;
00153         delete[] this->process_flag;
00154     } catch (...)  {
00155         // Suppress all exceptions during cleanup
00156     }
00157 }
```

Here is the call graph for this function:



3.1.3 Member Function Documentation

3.1.3.1 finish()

```
void ThreadPool::finish (
            bool secure = true )
```

Stops the thread pool.

Stops the thread pool and optionally joins worker threads.

Sets the exit flag and optionally waits for all worker threads to finish.

Parameters

secure	If true, joins all worker threads before returning.

Signals all threads to terminate and wakes them if they are waiting. If secure is enabled, this function blocks until all worker threads have finished execution.

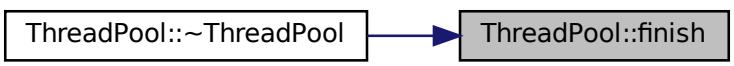
Parameters

secure	If true, waits for all threads to finish execution.

Definition at line 122 of file ThreadPool.cpp.

```
00122                                       {
00123     this->exit_flag = true;
00124
00125     unique_lock<mutex> lock(this->mtx);
00126     this->cv.notify_all();
00127     lock.unlock();
00128
00129     if (secure && this->state.load()) {
00130         for (unsigned int i = 0; i < this->num_threads; i++) {
00131             if (this->th[i].joinable()) {
00132                 try {
00133                     this->th[i].join();
00134                 } catch (...)  {
00135                     // Error while joining a worker thread
00136                 }
00137             }
00138         }
00139         this->state = false;
00140     }
00141 }
```

Here is the caller graph for this function:



3.1.3.2 length()

```
unsigned int ThreadPool::length ( )
```

Returns the number of pending tasks in the queue.

Returns

Number of tasks currently in the queue.

Number of tasks currently stored in the queue.

Definition at line 109 of file ThreadPool.cpp.

```
00109                                {
00110     return this->q.size();
00111 }
```

3.1.3.3 status()

```
bool ThreadPool::status ( )
```

Returns the current state of the thread pool.

Checks whether the thread pool is currently running.

Returns

True if the pool is running, false otherwise.

True if the pool is active, false otherwise.

Definition at line 100 of file ThreadPool.cpp.

```
00100                          {
00101     return this->state;
00102 }
```

3.1.3.4 submit()

```
template<class Func , class...  Args>
void ThreadPool::submit (
            Func && f,
            Args &&...  args )
```

Submits a task to the thread pool.

Adds a new task to the task queue.

The task is stored in the internal queue and will be executed by one of the worker threads.

Template Parameters

Func	Type of the callable object.
Args	Types of the arguments passed to the callable.

Parameters

f	Function to execute.
args	Arguments passed to the function.

Locks the mutex, pushes the task into the queue, and notifies one waiting worker thread.

Definition at line 148 of file ThreadPool.hpp.

```
00148                                                   {
00149
00150     lock_guard<mutex> lock(this->mtx);
00151
00152     this->q.push(
00153         move(
00154             [F = forward<Func>(f), ...args2 = forward<Args>(args)]() mutable {
00155                 F(args2...);
00156             }
00157         )
00158     );
00159
00160     this->cv.notify_one();
00161 }
```

3.1.3.5 wait()

```
void ThreadPool::wait ( )
```

Waits until the task queue becomes empty.

Waits until all queued tasks have been processed.

Note

This function performs a busy wait.

Blocks the calling thread until the task queue is empty and no worker thread is currently executing a task.

Definition at line 165 of file ThreadPool.cpp.

```
00165                       {
00166     unique_lock<mutex> lock(this->mtx);
00167
00168     this->waiting = true;
00169
00170     this->cv2.wait(lock, [&]() {
00171
00172         bool finished = true;
00173
00174         if (this->q.empty()) {
00175             for (unsigned int i = 0; i < this->num_threads; i++) {
00176                 if (this->process_flag[i].load()) {
00177                     finished = false;
00178                     break;
00179                 }
00180             }
00181             return finished;
00182         }
00183         return false;
00184     });
00185
00186     this->waiting = false;
00187 }
```

3.1.4 Field Documentation

3.1.4.1 cv

```
condition_variable ThreadPool::cv  [private]
```

Condition variable used to block threads until a new task is available

Definition at line 63 of file ThreadPool.hpp.

3.1.4.2 cv2

```
condition_variable ThreadPool::cv2  [private]
```

Condition variable used to wait until all tasks in the queue are completed

Definition at line 66 of file ThreadPool.hpp.

3.1.4.3 error

```
atomic<bool> ThreadPool::error = false
```

Indicates whether any worker thread has thrown an exception.

Definition at line 138 of file ThreadPool.hpp.

3.1.4.4 exit_flag

```
atomic<bool> ThreadPool::exit_flag  [private]
```

Flag indicating when threads should terminate

Definition at line 51 of file ThreadPool.hpp.

3.1.4.5 mtx

```
mutex ThreadPool::mtx  [private]
```

Mutex that protects access to the task queue

Definition at line 42 of file ThreadPool.hpp.

3.1.4.6 num_threads

```
unsigned int ThreadPool::num_threads  [private]
```

Number of threads managed by the pool

Definition at line 48 of file ThreadPool.hpp.

3.1.4.7 process_flag

```
atomic<bool>* ThreadPool::process_flag  [private]
```

Array of flags indicating whether each thread is executing a task

Definition at line 54 of file ThreadPool.hpp.

3.1.4.8 q

```
queue<function<void()> > ThreadPool::q  [private]
```

Queue that stores pending tasks

Definition at line 45 of file ThreadPool.hpp.

3.1.4.9 state

```
atomic<bool> ThreadPool::state  [private]
```

Indicates whether the thread pool is currently running

Definition at line 57 of file ThreadPool.hpp.

3.1.4.10 th

```
thread* ThreadPool::th  [private]
```

Pointer to the array of worker threads

Definition at line 39 of file ThreadPool.hpp.

3.1.4.11 waiting

```
atomic<bool> ThreadPool::waiting  [private]
```

Indicates whether the pool is waiting for all tasks to finish

Definition at line 60 of file ThreadPool.hpp.

The documentation for this class was generated from the following files:

- ThreadPool.hpp
- ThreadPool.cpp

Chapter 4

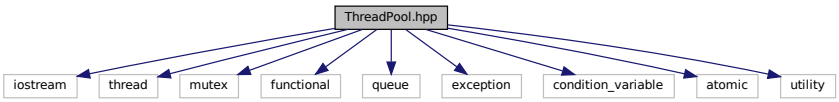
File Documentation

4.1 ThreadPool.hpp File Reference

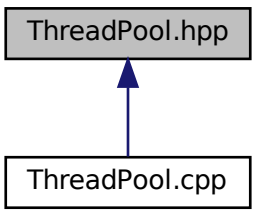
Declaration of a simple ThreadPool class.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <functional>
#include <queue>
#include <exception>
#include <condition_variable>
#include <atomic>
#include <utility>
```

Include dependency graph for ThreadPool.hpp:



This graph shows which files directly or indirectly include this file:



Data Structures

- class ThreadPool

  Manages a pool of worker threads to execute tasks concurrently.

4.1.1 Detailed Description

Declaration of a simple ThreadPool class.

Author

qwert-asci

This file contains the declaration of a basic thread pool that executes tasks using a fixed number of worker threads.

Definition in file ThreadPool.hpp.

4.2 ThreadPool.hpp

Go to the documentation of this file.

```
00001 /**
00002 * @file ThreadPool.hpp
00003 * @author qwert-asci
00004 * @brief Declaration of a simple ThreadPool class.
00005 *
00006 * This file contains the declaration of a basic thread pool
00007 * that executes tasks using a fixed number of worker threads.
00008 */
00009
00010 #ifndef THREADPOOL_HPP
00011 #define THREADPOOL_HPP
00012
00013 #include <iostream>
00014 #include <thread>
00015 #include <mutex>
00016 #include <functional>
00017 #include <queue>
00018 #include <exception>
00019 #include <condition_variable>
00020 #include <atomic>
00021 #include <utility>
00022
00023 using namespace std;
00024
00025 /**
00026 * @class ThreadPool
00027 * @brief Manages a pool of worker threads to execute tasks concurrently.
00028 *
00029 * The ThreadPool maintains a queue of tasks.  Worker threads continuously
00030 * fetch and execute tasks from this queue until the pool is stopped.
00031 *
00032 * @note Tasks have no return value.
00033 */
00034 class ThreadPool {
00035
00036 private:
00037
00038 /** Pointer to the array of worker threads */
00039     thread* th;
00040
00041 /** Mutex that protects access to the task queue */
00042     mutex mtx;
00043
00044 /** Queue that stores pending tasks */
00045     queue<function<void()>> q;
00046
00047 /** Number of threads managed by the pool */
00048     unsigned int num_threads;
00049
```

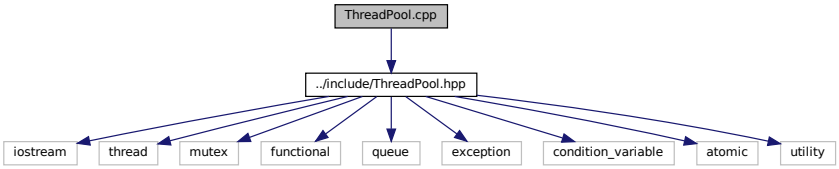
```
00050 /** Flag indicating when threads should terminate */
00051     atomic<bool> exit_flag;
00052
00053 /** Array of flags indicating whether each thread is executing a task */
00054     atomic<bool>* process_flag;
00055
00056 /** Indicates whether the thread pool is currently running */
00057     atomic<bool> state;
00058
00059 /** Indicates whether the pool is waiting for all tasks to finish */
00060     atomic<bool> waiting;
00061
00062 /** Condition variable used to block threads until a new task is available */
00063     condition_variable cv;
00064
00065 /** Condition variable used to wait until all tasks in the queue are completed */
00066     condition_variable cv2;
00067
00068 public:
00069
00070 /**
00071 * @brief Deleted default constructor.
00072 *
00073 * Forces the user to specify the number of threads.
00074 */
00075     ThreadPool() = delete;
00076
00077 /**
00078 * @brief Constructs a ThreadPool with a fixed number of worker threads.
00079 *
00080 * @param num_threads Number of worker threads to create.
00081 */
00082     explicit ThreadPool(unsigned int num_threads);
00083
00084 /**
00085 * @brief Submits a task to the thread pool.
00086 *
00087 * The task is stored in the internal queue and will be executed
00088 * by one of the worker threads.
00089 *
00090 * @tparam Func Type of the callable object.
00091 * @tparam Args Types of the arguments passed to the callable.
00092 * @param f Function to execute.
00093 * @param args Arguments passed to the function.
00094 */
00095     template <class Func, class...  Args>
00096     void submit(Func&& f, Args&&...  args);
00097
00098 /**
00099 * @brief Stops the thread pool.
00100 *
00101 * Sets the exit flag and optionally waits for all worker threads to finish.
00102 *
00103 * @param secure If true, joins all worker threads before returning.
00104 */
00105     void finish(bool secure = true);
00106
00107 /**
00108 * @brief Returns the current state of the thread pool.
00109 *
00110 * @return True if the pool is running, false otherwise.
00111 */
00112     bool status();
00113
00114 /**
00115 * @brief Returns the number of pending tasks in the queue.
00116 *
00117 * @return Number of tasks currently in the queue.
00118 */
00119     unsigned int length();
00120
00121 /**
00122 * @brief Waits until the task queue becomes empty.
00123 *
00124 * @note This function performs a busy wait.
00125 */
00126     void wait();
00127
00128 /**
00129 * @brief Destructor.
00130 *
00131 * Stops the thread pool and releases all allocated resources.
00132 */
00133     ~ThreadPool();
00134
00135 /**
00136 * @brief Indicates whether any worker thread has thrown an exception.
```

```
00137 */
00138     atomic<bool> error = false;
00139 };
00140
00141 /**
00142 * @brief Adds a new task to the task queue.
00143 *
00144 * Locks the mutex, pushes the task into the queue,
00145 * and notifies one waiting worker thread.
00146 */
00147 template <class Func, class...  Args>
00148 void ThreadPool::submit(Func&& f, Args&&...  args) {
00149
00150     lock_guard<mutex> lock(this->mtx);
00151
00152     this->q.push(
00153         move(
00154             [F = forward<Func>(f), ...args2 = forward<Args>(args)]() mutable {
00155                 F(args2...);
00156             }
00157         )
00158     );
00159
00160     this->cv.notify_one();
00161 }
00162
00163 #endif
00164
```

4.3 ThreadPool.cpp File Reference

Implementation of the ThreadPool class.

`#include "../include/ThreadPool.hpp"`
Include dependency graph for ThreadPool.cpp:



4.3.1 Detailed Description

Implementation of the ThreadPool class.

Definition in file ThreadPool.cpp.

4.4 ThreadPool.cpp

Go to the documentation of this file.

```
00001 /**
00002 * @file ThreadPool.cpp
00003 * @brief Implementation of the ThreadPool class.
00004 */
00005
00006 #include "../include/ThreadPool.hpp"
00007
00008 /**
```

```
00009 * @brief Constructs the thread pool and starts the worker threads.
00010 *
00011 * Each worker thread runs a loop that retrieves tasks
00012 * from the queue and executes them until termination
00013 * is requested.
00014 *
00015 * @param num_threads Number of worker threads to create.
00016 */
00017 ThreadPool::ThreadPool(unsigned int num_threads)
00018     :  num_threads(num_threads) {
00019
00020 /**
00021 * @brief Worker thread routine.
00022 *
00023 * Each worker thread continuously waits for tasks
00024 * in the queue and executes them while the pool
00025 * remains active and no exit has been requested.
00026 *
00027 * @param process_number Index of the worker thread.
00028 */
00029     function<void(unsigned int)> f = [&](unsigned int process_number) {
00030
00031         function<void()> task;
00032
00033         unique_lock<mutex> lock(this->mtx);
00034
00035         while (!this->exit_flag.load() && this->state.load()) {
00036
00037             cv.wait(lock, [&]() {
00038                 return !this->q.empty()
00039                        || this->exit_flag.load()
00040                        || !this->state.load();
00041             });
00042
00043             if (this->exit_flag.load() || !this->state.load()) {
00044                 lock.unlock();
00045                 break;
00046             }
00047
00048             task = move(this->q.front());
00049             this->q.pop();
00050
00051             lock.unlock();
00052
00053             this->process_flag[process_number] = true;
00054             try {
00055                 task();
00056             } catch (...)  {
00057                 this->error = true;
00058             }
00059             this->process_flag[process_number] = false;
00060
00061             lock.lock();
00062             if (this->waiting.load()) {
00063                 this->cv2.notify_one();
00064             }
00065         }
00066
00067         if (lock.owns_lock()) {
00068             lock.unlock();
00069         }
00070     };
00071
00072     this->exit_flag = false;
00073     this->waiting = false;
00074
00075     this->th = new (nothrow) thread[num_threads];
00076     this->process_flag = new (nothrow) atomic<bool>[num_threads];
00077
00078     if (this->th != nullptr && this->process_flag != nullptr) {
00079         this->state = true;
00080
00081         try {
00082             for (unsigned int i = 0; i < this->num_threads; i++) {
00083                 this->th[i] = thread(f, i);
00084                 this->process_flag[i] = false;
00085             }
00086         } catch (exception&) {
00087             this->state = false;
00088             // Failed to start one or more worker threads
00089         }
00090     } else {
00091         this->state = false;
00092     }
00093 }
00094
00095 /**
```

```
00096 * @brief Checks whether the thread pool is currently running.
00097 *
00098 * @return True if the pool is active, false otherwise.
00099 */
00100 bool ThreadPool::status() {
00101     return this->state;
00102 }
00103
00104 /**
00105 * @brief Returns the number of pending tasks in the queue.
00106 *
00107 * @return Number of tasks currently stored in the queue.
00108 */
00109 unsigned int ThreadPool::length() {
00110     return this->q.size();
00111 }
00112
00113 /**
00114 * @brief Stops the thread pool and optionally joins worker threads.
00115 *
00116 * Signals all threads to terminate and wakes them if they are waiting.
00117 * If secure is enabled, this function blocks until all worker threads
00118 * have finished execution.
00119 *
00120 * @param secure If true, waits for all threads to finish execution.
00121 */
00122 void ThreadPool::finish(bool secure) {
00123     this->exit_flag = true;
00124
00125     unique_lock<mutex> lock(this->mtx);
00126     this->cv.notify_all();
00127     lock.unlock();
00128
00129     if (secure && this->state.load()) {
00130         for (unsigned int i = 0; i < this->num_threads; i++) {
00131             if (this->th[i].joinable()) {
00132                 try {
00133                     this->th[i].join();
00134                 } catch (...)  {
00135                     // Error while joining a worker thread
00136                 }
00137             }
00138         }
00139         this->state = false;
00140     }
00141 }
00142
00143 /**
00144 * @brief Destructor.
00145 *
00146 * Ensures that the thread pool is stopped and all
00147 * allocated resources are released.
00148 */
00149 ThreadPool::~ThreadPool() {
00150     this->finish();
00151     try {
00152         delete[] this->th;
00153         delete[] this->process_flag;
00154     } catch (...)  {
00155         // Suppress all exceptions during cleanup
00156     }
00157 }
00158
00159 /**
00160 * @brief Waits until all queued tasks have been processed.
00161 *
00162 * Blocks the calling thread until the task queue is empty
00163 * and no worker thread is currently executing a task.
00164 */
00165 void ThreadPool::wait() {
00166     unique_lock<mutex> lock(this->mtx);
00167
00168     this->waiting = true;
00169
00170     this->cv2.wait(lock, [&]() {
00171
00172         bool finished = true;
00173
00174         if (this->q.empty()) {
00175             for (unsigned int i = 0; i < this->num_threads; i++) {
00176                 if (this->process_flag[i].load()) {
00177                     finished = false;
00178                     break;
00179                 }
00180             }
00181             return finished;
00182         }
```

```
00183         return false;
00184     });
00185 
00186     this->waiting = false;
00187 }
```

Index