

ThreadPool

Generated by Doxygen 1.9.4

1 Data Structure Index	1
1.1 Data Structures	1
2 File Index	3
2.1 File List	3
3 Data Structure Documentation	5
3.1 ThreadPool Class Reference	5
3.1.1 Detailed Description	6
3.1.2 Constructor & Destructor Documentation	6
3.1.2.1 ThreadPool() [1/2]	6
3.1.2.2 ThreadPool() [2/2]	6
3.1.2.3 ~ThreadPool()	8
3.1.3 Member Function Documentation	8
3.1.3.1 finish()	8
3.1.3.2 length()	9
3.1.3.3 status()	9
3.1.3.4 submit()	10
3.1.3.5 wait()	10
3.1.4 Field Documentation	11
3.1.4.1 cv	11
3.1.4.2 cv2	11
3.1.4.3 error	11
3.1.4.4 exit_flag	12
3.1.4.5 mtx	12
3.1.4.6 num_threads	12
3.1.4.7 process_flag	12
3.1.4.8 q	13
3.1.4.9 state	13
3.1.4.10 th	13
3.1.4.11 waiting	13
4 File Documentation	15
4.1 ThreadPool.hpp File Reference	15
4.1.1 Detailed Description	16
4.2 ThreadPool.hpp	16
4.3 ThreadPool.cpp File Reference	18
4.3.1 Detailed Description	18
4.4 ThreadPool.cpp	19
Index	23

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

ThreadPool	Manages a pool of worker threads to execute tasks concurrently	5
----------------------------	--	---

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

ThreadPool.hpp	Declaration of a simple <code>ThreadPool</code> class	15
ThreadPool.cpp	Implementation of the <code>ThreadPool</code> class	18

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 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 process in queue.
- **void wait ()**
Waits until the task queue becomes empty.
- **~ThreadPool ()**
Destructor.

Data Fields

- **atomic< bool > error =false**

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 35 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 threads.

Constructs and starts the worker threads.

Parameters

<code>num_threads</code>	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.
Worker thread function.

Continuously checks the task queue and executes tasks while the pool is active and the exit flag is not set.

Definition at line 15 of file [ThreadPool.cpp](#).

```

00016     : num_threads(num_threads) {
00017
00018     /**
00019      * @brief Worker thread function.
00020      *
00021      * Continuously checks the task queue and executes tasks
00022      * while the pool is active and the exit flag is not set.
00023     */
00024     function<void(unsigned int)> f = [&](unsigned int process_number) {
00025
00026         function<void()> task;
00027
00028         unique_lock<mutex> lock(this->mtx);
00029
00030         while (!this->exit_flag.load() && this->state.load()) {
00031
00032             cv.wait(lock, [&] () {
00033                 return !this->q.empty() || (this->exit_flag.load() || !this->state.load());
00034             });
00035         };
00036
00037
00038
00039         if(this->exit_flag.load() || !this->state.load()){
00040             lock.unlock();
00041             break;
00042         }
00043
00044         task = move(this->q.front());
00045         this->q.pop();
00046
00047         lock.unlock();
00048
00049
00050         this->process_flag[process_number]=true;
00051         try{
00052             task();
00053         }catch(...){
00054             this->error=true;
00055         }
00056         this->process_flag[process_number]=false;
00057         if(this->waiting.load()==true){
00058             this->cv2.notify_one();
00059         }
00060         lock.lock();
00061
00062     };
00063 };
00064
00065 this->exit_flag = false;
00066 this->waiting=false;
00067
00068 this->th = new (nothrow) thread[num_threads];
00069 this->process_flag = new (nothrow) atomic<bool>[num_threads];
00070
00071 if (this->th != nullptr && this->process_flag != nullptr) {
00072     this->state = true;
00073
00074     try {
00075         for (unsigned int i = 0; i < this->num_threads; i++) {
00076             this->th[i] = thread(f,i);
00077             this->process_flag[i]=false;
00078         }
00079     } catch (exception& e) {
00080         this->state = false;
00081         // cout << "The system is unable to start a new thread" << endl;
00082     }
00083 } else {
00084     this->state = false;
00085 }
00086 }
```

3.1.2.3 ~ThreadPool()

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

Destructor.

Stops the pool and releases all allocated resources.

Ensures that the pool is stopped and memory is released.

Definition at line 141 of file [ThreadPool.cpp](#).

```
00141     {
00142         this->finish();
00143     try{
00144         delete[] this->th;
00145         delete[] this->process_flag;
00146     }catch(...){
00147     }
00148 }
00149 }
```

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 threads.

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

Parameters

<code>secure</code>	If true, joins all worker threads.
<code>secure</code>	If true, waits for all threads to finish execution.

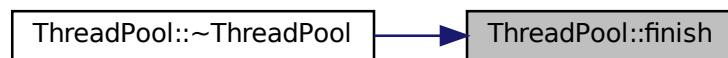
Definition at line 116 of file [ThreadPool.cpp](#).

```
00116 {
```

```

00117     this->exit_flag = true;
00118
00119     this->cv.notify_all();
00120
00121     if (secure && this->state.load()) {
00122         for (unsigned int i = 0; i < this->num_threads; i++) {
00123             if (this->th[i].joinable()) {
00124                 try {
00125                     this->th[i].join();
00126                 } catch (...) {
00127                     // cout << "Error joining thread " << i << endl;
00128                 }
00129             }
00130         }
00131     }
00132     this->state = false;
00133 }
00134 }
```

Here is the caller graph for this function:



3.1.3.2 length()

unsigned int ThreadPool::length ()

Returns the number of process in queue.

Provides the number of process in queue.

Returns

integer with the length of the queue.

Definition at line 105 of file [ThreadPool.cpp](#).

```

00105
00106     return this->q.size();
00107 }
```

3.1.3.3 status()

bool ThreadPool::status ()

Returns the current state of the thread pool.

Checks whether the thread pool is running.

Returns

True if the pool is running, false otherwise.

True if the pool is active, false otherwise.

Definition at line 93 of file [ThreadPool.cpp](#).

```

00093
00094     return this->state;
00095 }
```

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

<i>Func</i>	Callable object type.
<i>Args</i>	Argument types for the callable.

Parameters

<i>f</i>	Function to execute.
<i>args</i>	Arguments passed to the function.

Locks the mutex, pushes the task into the queue, and then unlocks the mutex.

Definition at line 153 of file [ThreadPool.hpp](#).

```
00153
00154
00155
00156     {
00157         lock_guard<mutex> lock(this->mtx);
00158
00159
00160         this->q.push(
00161             move(
00162                 [&] () {
00163                     f(args...);
00164                 }
00165             )
00166         );
00167     }
00168     this->cv.notify_one();
00169 }
```

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.

Definition at line 154 of file [ThreadPool.cpp](#).

```

00154         {
00155             unique_lock<mutex> lock(this->mtx);
00156
00157             this->waiting=true;
00158
00159             this->cv2.wait(lock,[&](){
00160
00161                 bool flag=true;
00162
00163                 if(this->q.empty()){
00164
00165                     for(unsigned int i=0;i<this->num_threads;i++){
00166                         if(this->process_flag[i].load()){
00167                             flag=false;
00168                             break;
00169                         }
00170                     }
00171
00172                     return flag?true:false;
00173                 }else{
00174                     return false;
00175                 }
00176             });
00177
00178             this->waiting=false;
00180     }
00181 }
```

3.1.4 Field Documentation

3.1.4.1 cv

condition_variable ThreadPool::cv [private]

Condition variable used to stop the threads until a new process is pushed

Definition at line 64 of file [ThreadPool.hpp](#).

3.1.4.2 cv2

condition_variable ThreadPool::cv2 [private]

Condition variable used to wait until all the queue finish

Definition at line 67 of file [ThreadPool.hpp](#).

3.1.4.3 error

atomic<bool> ThreadPool::error =false

Parameters

<i>Flag</i>	that indicates if any thread has thrown an exception
-------------	--

Definition at line 143 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 52 of file [ThreadPool.hpp](#).

3.1.4.5 mtx

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

Mutex protecting access to the task queue

Definition at line 43 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 49 of file [ThreadPool.hpp](#).

3.1.4.7 process_flag

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

Array of flags indicating if a thread is running a process or not

Definition at line 55 of file [ThreadPool.hpp](#).

3.1.4.8 q

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

Queue containing pending tasks

Definition at line 46 of file [ThreadPool.hpp](#).

3.1.4.9 state

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

Indicates whether the thread pool is active

Definition at line 58 of file [ThreadPool.hpp](#).

3.1.4.10 th

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

Pointer to the array of worker threads

Definition at line 40 of file [ThreadPool.hpp](#).

3.1.4.11 waiting

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

Indicates whether the thread pool is waiting to finish all the tasks or not

Definition at line 61 of file [ThreadPool.hpp](#).

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

- [ThreadPool.hpp](#)
- [ThreadPool.cpp](#)

Chapter 4

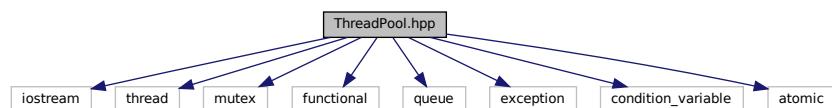
File Documentation

4.1 ThreadPool.hpp File Reference

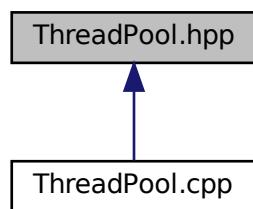
Declaration of a simple [ThreadPool](#) class.

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

Include 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
00011 #ifndef THREADPOOL_HPP
00012 #define THREADPOOL_HPP
00013
00014 #include <iostream>
00015 #include <thread>
00016 #include <mutex>
00017 #include <functional>
00018 #include <queue>
00019 #include <exception>
00020 #include <condition_variable>
00021 #include <atomic>
00022
00023
00024 using namespace std;
00025
00026 /**
00027 * @class ThreadPool
00028 * @brief Manages a pool of worker threads to execute tasks concurrently.
00029 *
00030 * The ThreadPool maintains a queue of tasks. Worker threads continuously
00031 * fetch and execute tasks from this queue until the pool is stopped.
00032 *
00033 * @note Tasks have no return value.
00034 */
00035 class ThreadPool {
00036
00037 private:
00038
00039 /** Pointer to the array of worker threads */
00040     thread* th;
00041
00042 /** Mutex protecting access to the task queue */
00043     mutex mtx;
00044
00045 /** Queue containing pending tasks */
00046     queue<function<void()>> q;
00047
00048 /** Number of threads managed by the pool */
00049     unsigned int num_threads;
```

```
00050
00051     /** Flag indicating when threads should terminate */
00052     atomic<bool> exit_flag;
00053
00054     /** Array of flags indicating if a thread is running a process or not */
00055     atomic<bool> * process_flag;
00056
00057     /** Indicates whether the thread pool is active */
00058     atomic<bool> state;
00059
00060     /** Indicates whether the thread pool is waiting to finish all the tasks or not*/
00061     atomic<bool> waiting;
00062
00063     /** Condition variable used to stop the threads until a new process is pushed */
00064     condition_variable cv;
00065
00066     /** Condition variable used to wait until all the queue finish */
00067     condition_variable cv2;
00068
00069 public:
00070
00071     /**
00072      * @brief Deleted default constructor.
00073      *
00074      * Forces the user to specify the number of threads.
00075      */
00076     ThreadPool() = delete;
00077
00078     /**
00079      * @brief Constructs a ThreadPool with a fixed number of threads.
00080      *
00081      * @param num_threads Number of worker threads to create.
00082      */
00083     explicit ThreadPool(unsigned int num_threads);
00084
00085     /**
00086      * @brief Submits a task to the thread pool.
00087      *
00088      * The task is stored in the internal queue and will be executed
00089      * by one of the worker threads.
00090      *
00091      * @tparam Func Callable object type.
00092      * @tparam Args Argument types for the callable.
00093      * @param f Function to execute.
00094      * @param args Arguments passed to the function.
00095      *
00096      */
00097     template <class Func, class... Args>
00098     void submit(Func&& f, Args&&... args);
00099
00100    /**
00101     * @brief Stops the thread pool.
00102     *
00103     * Sets the exit flag and optionally waits for all threads to finish.
00104     *
00105     * @param secure If true, joins all worker threads.
00106     */
00107     void finish(bool secure = true);
00108
00109    /**
00110     * @brief Returns the current state of the thread pool.
00111     *
00112     * @return True if the pool is running, false otherwise.
00113     */
00114     bool status();
00115
00116
00117    /**
00118     * @brief Returns the number of process in queue.
00119     *
00120     * @return integer with the length of the queue.
00121     */
00122     unsigned int length();
00123
00124    /**
00125     * @brief Waits until the task queue becomes empty.
00126     *
00127     * @note This function performs a busy wait.
00128     */
00129     void wait();
00130
00131    /**
00132     * @brief Destructor.
00133     *
00134     * Stops the pool and releases all allocated resources.
00135     */
00136     ~ThreadPool();
```

```

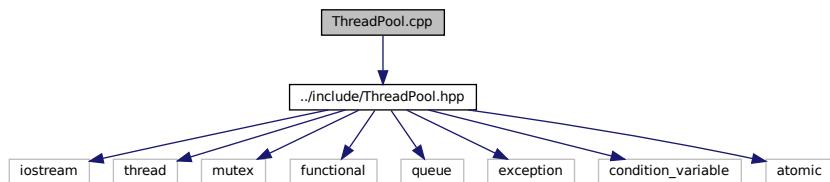
00137
00138
00139 /**
00140 * @param Flag that indicates if any thread has thrown an exception
00141 *
00142 */
00143     atomic<bool> error=false;
00144 };
00145
00146 /**
00147 * @brief Adds a new task to the task queue.
00148 *
00149 * Locks the mutex, pushes the task into the queue,
00150 * and then unlocks the mutex.
00151 */
00152 template <class Func, class... Args>
00153 void ThreadPool::submit(Func&& f, Args&&... args) {
00154
00155
00156 {
00157     lock_guard<mutex> lock(this->mtx);
00158
00159     this->q.push(
00160         move(
00161             [&] () {
00162                 f(args...);
00163             }
00164         );
00165     );
00166 }
00167
00168     this->cv.notify_one();
00169 }
00170 }
00171
00172 #endif

```

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 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 ThreadPool::ThreadPool(unsigned int num_threads)
00016     : num_threads(num_threads) {
00017
00018 /**
00019 * @brief Worker thread function.
00020 *
00021 * Continuously checks the task queue and executes tasks
00022 * while the pool is active and the exit flag is not set.
00023 */
00024     function<void(unsigned int)> f = [&](unsigned int process_number) {
00025
00026         function<void()> task;
00027
00028         unique_lock<mutex> lock(this->mtx);
00029
00030         while (!this->exit_flag.load() && this->state.load()) {
00031
00032             cv.wait(lock, [&](){
00033                 return !this->q.empty() || (this->exit_flag.load() || !this->state.load());
00034             });
00035         );
00036
00037
00038
00039
00040         if(this->exit_flag.load() || !this->state.load()){
00041             lock.unlock();
00042             break;
00043         }
00044
00045         task = move(this->q.front());
00046         this->q.pop();
00047
00048         lock.unlock();
00049
00050
00051         this->process_flag[process_number]=true;
00052         try{
00053             task();
00054         }catch(...){
00055             this->error=true;
00056         }
00057         this->process_flag[process_number]=false;
00058         if(this->waiting.load()==true){
00059             this->cv2.notify_one();
00060         }
00061         lock.lock();
00062     };
00063 };
00064
00065 this->exit_flag = false;
00066 this->waiting=false;
00067
00068 this->th = new (nothrow) thread[num_threads];
00069 this->process_flag = new (nothrow) atomic<bool>[num_threads];
00070
00071 if (this->th != nullptr && this->process_flag != nullptr) {
00072     this->state = true;
00073
00074     try {
00075         for (unsigned int i = 0; i < this->num_threads; i++) {
00076             this->th[i] = thread(f,i);
00077             this->process_flag[i]=false;
00078         }
00079     } catch (exception& e) {
00080         this->state = false;
00081         // cout << "The system is unable to start a new thread" << endl;
00082     }

```

```
00083     } else {
00084         this->state = false;
00085     }
00086 }
00087
00088 /**
00089 * @brief Checks whether the thread pool is running.
00090 *
00091 * @return True if the pool is active, false otherwise.
00092 */
00093 bool ThreadPool::status() {
00094     return this->state;
00095 }
00096
00097
00098
00099
00100 /**
00101 * @brief Provides the number of process in queue.
00102 *
00103 * @return integer with the length of the queue.
00104 */
00105 unsigned int ThreadPool::length() {
00106     return this->q.size();
00107 }
00108
00109
00110
00111 /**
00112 * @brief Stops the thread pool and optionally joins threads.
00113 *
00114 * @param secure If true, waits for all threads to finish execution.
00115 */
00116 void ThreadPool::finish(bool secure) {
00117     this->exit_flag = true;
00118
00119     this->cv.notify_all();
00120
00121     if (secure && this->state.load()) {
00122         for (unsigned int i = 0; i < this->num_threads; i++) {
00123             if (this->th[i].joinable()) {
00124                 try {
00125                     this->th[i].join();
00126                 } catch (...) {
00127                     // cout << "Error joining thread " << i << endl;
00128                 }
00129             }
00130         }
00131     }
00132     this->state = false;
00133 }
00134 }
00135
00136 /**
00137 * @brief Destructor.
00138 *
00139 * Ensures that the pool is stopped and memory is released.
00140 */
00141 ThreadPool::~ThreadPool() {
00142     this->finish();
00143     try{
00144         delete[] this->th;
00145         delete[] this->process_flag;
00146     }catch(...){}
00147
00148 }
00149 }
00150
00151 /**
00152 * @brief Waits until all queued tasks have been processed.
00153 */
00154 void ThreadPool::wait() {
00155     unique_lock<mutex> lock(this->mtx);
00156
00157     this->waiting=true;
00158
00159     this->cv2.wait(lock,[&](){
00160
00161         bool flag=true;
00162
00163         if(this->q.empty()){
00164
00165             for(unsigned int i=0;i<this->num_threads;i++){
00166                 if(this->process_flag[i].load()){
00167                     flag=false;
00168                     break;
00169                 }
00170             }
00171         }
00172     });
00173 }
```

```
00170         }
00171     return flag?true:false;
00172 }else{
00173     return false;
00174 }
00175 }
00176 );
00177 );
00178 this->waiting=false;
00180
00181 }
```


Index

- ~ThreadPool
 - ThreadPool, 7
- cv
 - ThreadPool, 11
- cv2
 - ThreadPool, 11
- error
 - ThreadPool, 11
- exit_flag
 - ThreadPool, 12
- finish
 - ThreadPool, 8
- length
 - ThreadPool, 9
- mtx
 - ThreadPool, 12
- num_threads
 - ThreadPool, 12
- process_flag
 - ThreadPool, 12
- q
 - ThreadPool, 12
- state
 - ThreadPool, 13
- status
 - ThreadPool, 9
- submit
 - ThreadPool, 9
- th
 - ThreadPool, 13
- ThreadPool, 5
 - ~ThreadPool, 7
 - cv, 11
 - cv2, 11
 - error, 11
 - exit_flag, 12
 - finish, 8
 - length, 9
 - mtx, 12
 - num_threads, 12
 - process_flag, 12
- q, 12
- state, 13
- status, 9
- submit, 9
- th, 13
- ThreadPool, 6
- wait, 10
- waiting, 13
- ThreadPool.cpp, 18, 19
- ThreadPool.hpp, 15, 16
- wait
 - ThreadPool, 10
- waiting
 - ThreadPool, 13