C++ Concurrency

future<RetType> ( ) M M=

RetType | RetType& | void get()
get the result - blocks until result is ready; return type determined by *RetType* template parameter

bool valid()
true if *get* hasn't been called

shared_future<RetType> share()
convert *future* to *shared_future*

void wait()
block until result is available

future_status wait_for(const duration&)
wait for the result for a specified period; unblock when result is available or after duration elapsed

future_status wait_until(const time_point&)
wait for the result until the specfied point in time; unblock when result is available or when time point passed

shared_future<RetType>

( ) C C= M M=

shared_future(future<RetType>&&)
move-construct from a *future*

RetType | RetType& | void get()
get the result - blocks until result is ready; return type determined by *RetType* template parameter

bool valid()
true if *get* hasn't been called

shared_future<RetType> share()
convert *future* to *shared_future*

void wait()
block until result is available

future_status wait_for(const duration&)
wait for the result for a specified period; unblock when result is available or after duration

future_status wait_until(const time_point&)
wait for the result until the specfied point in time

Legend

( )	default constructor	C=	copy assignment operator
C	copy constructor	M=	move assignment operator
M	move constructor	⟷	swap method

cpprocks.com
@cpp_rocks

thread ( ) M M= ⟷

thread<F, Args...>(F&&, Args&&...)
construct from F and Args

bool joinable()
true if the thread hasn't been detached

void join()
block until thread completes

void detach()
give up control of the thread

id get_id()
get thread ID

native_handle_type native_handle()
get platform specific thread handle

static unsigned hardware_concurrency()
return an estimate of hardware thread contexts

this_thread *namespace*

thread::id get_id()
return the unique ID of the calling thread

void yield()
offer the implementation a chance to reschedule

void sleep_until(const time_point&)
block the calling thread until specified time

void sleep_for(const duration&)
block the calling thread for specified period

Free functions

future<RetTypeOfF> async([launch], F&&, Args&&...)
return a future and execute *F* with *Args* according to launch policy if provided, or with *launch::async* | *launch::deferred* otherwise

void lock<L1, L2, L3...>(L1&, L2&, L3&...)
lock all arguments using a deadlock avoidance algorithm; in case of failure, unlock all previously locked arguments and return

int try_lock<L1, L2, L3...>(L1&, L2&, L3&...)
call *try_lock* on each argument in order & return -1; if an argument can't be locked, unlock all previous arguments & return its index

void call_once(once_flag&, F&&, Args&&...)
execute *F* with *Args* only once in a multi-threaded context

lock_guard<Mutex>

lock_guard(Mutex&, [adopt_lock_t])
lock the mutex on construction and release on destruction

packaged_task<RetType, ArgTypes...> ( ) M M= ⟷

packaged_task<F>(F&&)
packaged_task<F, Alloc>(allocator_arg_t, const Alloc&, F&&)
construct from *F*, using *Alloc* for internal data structures (if provided)

future<RetType> get_future()
return a future for this task

void operator()(ArgTypes...)
execute the task and signal the future

bool valid()
true if the task has shared state

void make_ready_at_thread_exit(ArgTypes...)
execute the task and signal the future at thread exit

void reset()
construct new shared state, abandon old state

promise<RetType> ( ) M M= ⟷

promise<Alloc>(allocator_arg_t, const Alloc&)
construct using Alloc for shared state

future<RetType> get_future()
return a future for this promise

void set_value(const RetType&)
void set_value(RetType&& | RetType& | void)
set the result and signal the future

void set_exception(exception_ptr)
set an exception and signal the future

void set_value_at_thread_exit(const RetType&)
void set_value_at_thread_exit(RetType&& | RetType& | void)
set result and signal the future at thread exit

void set_exception_at_thread_exit (exception_ptr)
set exception and signal the future at thread exit

unique_lock<Mutex> ( ) M M= ⟷

unique_lock(Mutex&, [defer_lock_t | try_to_lock_t | adopt_lock_t])
possibly acquire mutex on construction

mutex_type* release()
unlock and return a pointer to mutex

bool owns_lock()
true if the mutex is locked

mutex_type* mutex()
return a pointer to mutex

Also has the same methods as *timed_mutex* (except *native_handle*)

condition_variable ( )

void notify_one()
unblock one of the waiting threads

void notify_all()
unblock all of the waiting threads

void wait(unique_lock<mutex>&, [Predicate])
unlock the mutex and block the thread until the condition variable is signalled; use *Predicate* to check for spurious wakeups

cv_status | bool wait_until (unique_lock<mutex>&, const time_point&, [Predicate])
like *wait*, but only wait until specified time point; return *cv_status* or, if *Predicate* is supplied, the value of *Predicate*

cv_status | bool wait_for (unique_lock<mutex>&, const duration&, [Predicate])
like *wait*, but only wait for the specified duration; return *cv_status* or, if *Predicate* is supplied, the value of *Predicate*

native_handle_type native_handle()
get platform specific handle

condition_variable_any ( )

Same interface as *condition_variable*, but *wait** methods allow a custom lock class in place of *unique_lock*, and *native_handle* method isn't available

mutex/recursive_mutex ( )

void lock()
recursive_mutex allows multiple calls to *lock* with increasing levels of ownership

bool try_lock()
immediately return *false* if unable to lock

void unlock()

native_handle_type native_handle()
get platform specific handle

timed_mutex/ recursive_timed_mutex ( )

Same as *mutex/recursive_mutex*, with two extra methods:

bool try_lock_for(const duration&)
try to lock for the specified duration

bool try_lock_until(const time_point&)
try to lock until the specified time point