CONCURRENCY

Bùi Tiến Lên

2022

Contents

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

Concurrency

Concept 1

Concurrency, the execution of several tasks simultaneously, is widely used

- to improve throughput (by using several processors for a single computation) or

- to improve responsiveness (by allowing one part of a program to progress while another is waiting for a response)

- The C++ standard-library support is primarily aimed at supporting systems-level concurrency rather than directly providing sophisticated higher-level concurrency models.

Tasks and Threads

Passing Arguments

Returning Results

Sharing Data

Waiting for Events

Communicating Tasks

Tasks vs Threads

Concept 2

- A *task* can be
    - a function
    - a function object
    - a lambda expression
- A *thread* is the system-level representation of a task in a program.
- A task to be executed concurrently with other tasks is launched by constructing a thread with the task as its argument.

- C++: Using #include <thread>

Tasks and Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

Tasks vs Threads (cont.)

```cpp
void func() {               // function
  cout << "task A\n";
}
class FuncClass {           // function object
public:
  void operator()() {
    cout << "Task B\n";
  }
};
void doSomething() {
    FuncClass funcObj;
    thread t1(func);
    thread t2(funcObj);
    thread t3([]() {cout << "task C\n"; });
    t1.join();             // wait for t1
    t2.join();             // wait for t2
    t3.join();             // wait for t3
}
```

Tasks and Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

`std::thread` Members

Member Name	Description
`joinable`	check if thread joinable
`get_id`	get ID of thread
`native_handle`	get native handle for thread
`hardware_concurrency`	get number of concurrent threads supported by hardware
`join`	wait for thread to finish executing
`detach`	permit thread to execute independently
`swap`	swap threads

**Tasks and
Threads**

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

The `std::this_thread` Namespace

Name	Description
get_id	get ID of current thread
yield	suggest rescheduling current thread so as to allow other threads to run
sleep_for	blocks execution of current thread for at least specified duration
sleep_until	blocks execution of current thread until specified time reached

Passing Arguments

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

Passing Arguments

- A task needs data to work upon.

```cpp
void func(vector<double>& v) {...} // function do something with v
    ...
}
class FuncClass {                  // function object do something with v
private:
    vector<double>& v;
public:
    FuncClass(vector<double>& vv):v(vv) {}
    void operator()() {...}        // application operator
};
int main() {
    vector<double> some_vec {1,2,3,4,5,6,7,8,9};
    vector<double> other_vec {10,11,12,13,14};
    FuncClass funcObj(other_vec);
    thread t1(f, ref(some_vec));   // executes in a separate thread
    thread t2(funcObj);            // executes in a separate thread
    t1.join();
    t2.join();
}
```

Returning Results

Tasks and
Threads

Passing
Arguments

**Returning
Results**

Sharing Data

Waiting for
Events

Communicating
Tasks

Returning Results

🧠

- Pass the input data by const reference
- Pass the location of a place to deposit the result as a separate argument

```
void func(const vector<double>& v, double* res) {
  *res = accumulate(v.begin(), v.end(), 0);
}
void doSomething() {
  vector<double> v(10000, 1);
  double res;
  thread t(func, v, &res);
  t.join();
  cout << res << endl;
}
```

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

Problem of Sharing Data

Concept 3

A **race condition** occurs when two or more threads can access shared data and they try to change it at the same time

```cpp
unsigned long long counter = 0;
void func() {
  for (unsigned long long i = 0; i < 1000000; ++i) {
    ++counter;
  }
}
int main() {
  thread t1(func);
  thread t2(func);
  t1.join();
  t2.join();
  cout << counter << endl;
}
```

14

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

Mutexes

Concept 4

A **mutex** is a locking mechanism used to synchronize access to a shared resource
by providing mutual exclusion.

- A mutex has two basic operations:
  - acquire: lock the mutex
  - release: unlock the mutex
- A mutex can be held by **only one thread** at any given time.

- C++: Using #include <mutex>

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

Mutexes (cont.)

```
mutex m;
unsigned long long counter = 0;
void func() {
  for (unsigned long long i = 0; i < 1000000; ++i) {
    m.lock();
    ++counter;
    m.unlock();
  }
}
int main() {
  thread t1(func);
  thread t2(func);
  t1.join();
  t2.join();
  cout << counter << endl;
}
```

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

Deadlock

🧠

Concept 5

A **deadlock** is a state in which each member of a group is waiting for another member, including itself, to take action.

Concept 6

A **livelock** is similar to a *deadlock*, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

Deadlock (cont.)

- Consider the following program

```
mutex m;
void recursiveFunction(int i) {
  m.lock();
  if(i > 0) {
    cout << i;
    recursiveFunction(i-1);
  }
  m.unlock();
}
int main() {
  recursiveFunction(1);
}
```

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

Reentrant mutex

Concept 7

A **reentrant mutex** (**recursive mutex**) is a particular type of mutual exclusion mutex that may be locked multiple times by the same process/thread, without causing a **deadlock**.

```
recursive_mutex m;
void recursiveFunction(int i) {
  m.lock();
  if(i > 0) {
    cout << i;
    recursiveFunction(i-1);
  }
  m.unlock();
}
int main() {
  recursiveFunction(1);
}
```

Tasks and
Threads
Passing
Arguments
Returning
Results
Sharing Data
Waiting for
Events
Communicating
Tasks

Use of resource handles

- Use of RAII resource handles, such as scoped_lock and unique_lock, is simpler and far safer than explicitly locking and unlocking mutex.

```
mutex m;
unsigned long long counter = 0;
void func() {
  for (unsigned long long i = 0; i < 1000000; ++i) {
    unique_lock<mutex> lck(m); // acquire mutex
    ++counter;
  } // release mutex implicitly
}
int main() {
  thread t1(func);
  thread t2(func);
  t1.join();
  t2.join();
  cout << counter << endl;
}
```

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

Shared mutexes

🦾

- The basic `mutex` allows one thread at a time to access data. One of the most common ways of sharing data is among many readers and a single writer. This "reader-writer lock" idiom is supported be `shared_mutex`.
- A reader will acquire the mutex "shared" so that other readers can still gain access, whereas a writer will demand exclusive access.

```
shared_mutex mx;              // a mutex that can be shared
void reader() {
    shared_lock<shared_mutex> lck(mx); // willing to share access
        with other
                              // readers
    // ... read ...
}
void writer() {
    unique_lock<shared_mutex> lck(mx); // needs exclusive (unique)
        access
    // ... write ...
}
```

Waiting for Events

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

Waiting for Events

- Sometimes, a `thread` needs to wait for some kind of external event, such as another `thread` completing a task or a certain amount of time having passed.
- The basic support for communicating using external events is provided by `condition_variable`.
- A `condition_variable` is a mechanism allowing one `thread` to wait for another.

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

**Waiting for
Events**

Communicating
Tasks

`condition_variable` Members

Name	Description
`notify_one`	notify one waiting thread
`notify_all`	notify all waiting threads
`wait`	blocks current thread until notified
`wait_for`	blocks current thread until notified or specified duration passed
`wait_until`	blocks current thread until notified or specified time point reached

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

Communicating
Tasks

Example

- Consider the classical example of two threads communicating by passing messages through a queue.

```cpp
class Message {      // object to be communicated
     // ...
};
queue<Message> mqueue;     // the queue of messages
condition_variable mcond;  // the variable communicating events
mutex mmutex;              // for synchronizing access to mcond
```

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

**Waiting for
Events**

Communicating
Tasks

Example (cont.)

```cpp
void consumer() {
  while (true) {
    unique_lock<mutex> lck(mmutex);  // acquire mmutex
    mcond.wait(lck, [] { return !mqueue.empty(); });
    // release lck and wait;
    // re-acquire lck upon wakeup
    // don't wake up unless mqueue is non-empty

    auto m = mqueue.front();  // get the message
    mqueue.pop();
    lck.unlock();             // release lck (optional)
    // ... process m ...
  }
}
```

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

**Waiting for
Events**

Communicating
Tasks

Example (cont.)

```
void producer() {
  while (true) {
    Message m;
    // ... fill the message ...
    unique_lock<mutex> lck(mmutex); // protect operations
    mqueue.push(m);
    mcond.notify_one();           // notify
  } // release lock (at end of scope)
}
```

Communicating Tasks

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

**Communicating
Tasks**

Communicating Tasks 🧠

- The standard library provides a few facilities to allow programmers to operate at the conceptual level of tasks (work to potentially be done concurrently) rather than directly at the lower level of threads and locks:
  - `future` and `promise` for returning a value from a task spawned on a separate thread
  - `async()` for launching of a task in a manner very similar to calling a function
- C++: `#include <future>`

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

**Communicating
Tasks**

promise **and** future

- The important point about future and promise is that they enable a transfer of a value between two tasks without explicit use of a lock
- promise Members

Name	Description
swap	swap two promise objects
get_future	get future associated with promised result
set_value	set result to specified value
set_value_at_thread_exit	set result to specified value while delivering notification only at thread exit
set_exception	set result to specified exception
set_exception_at_thread_exit	set result to specified exception while delivering notification only at thread exit

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

**Communicating
Tasks**

promise **and** future **(cont.)**

- future Members

Name	Description
share	transfer shared state to shared_future object
get	get result
valid	check if future object refers to shared state
wait	wait for result to become available
wait_for	wait for result to become available or time duration to expire
wait_until	wait for result to become available or time point to be reached

Tasks and
Threads
Passing
Arguments
Returning
Results
Sharing Data
Waiting for
Events
**Communicating
Tasks**

Example

```cpp
void f(promise<X>& px) {
    // a task: place the result in px
    // ...
    try {
        X res;
        // ... compute a value for res ...
        px.set_value(res);
    }
    catch (...) {
        // pass the exception to the future's thread
        px.set_exception(current_exception());
    }
}
```

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

**Communicating
Tasks**

Example (cont.)

```
void g(future <X>& fx) {
     // a task: get the result from fx
     // ...
     try {
          X v = fx.get();
          // if necessary, wait for the value to get computed
          // ... use v ...
     }
     catch (...) {
          // ... handle error ...
     }
}
```

Tasks and
Threads

Passing
Arguments

Returning
Results

Sharing Data

Waiting for
Events

**Communicating
Tasks**

async()

- To launch tasks to potentially run asynchronously, we can use async()

```cpp
double comp4(vector<double>& v) {
    // spawn many tasks if v is large enough
    if (v.size()<10000) // is it worth using concurrency?
        return accum(v.begin(),v.end(),0.0);

    auto v0 = &v[0];
    auto sz = v.size();

    auto f0 = async(accum,v0,v0+sz/4,0.0);        // first quarter
    auto f1 = async(accum,v0+sz/4,v0+sz/2,0.0);   // second quarter
    auto f2 = async(accum,v0+sz/2,v0+sz*3/4,0.0); // third quarter
    auto f3 = async(accum,v0+sz*3/4,v0+sz,0.0);   // fourth quarter

    // collect and combine the results
    return f0.get()+f1.get()+f2.get()+f3.get();
}
```

References

📄 Deitel, P. (2016).
C++: How to program.
Pearson.

📄 Gaddis, T. (2014).
Starting Out with C++ from Control Structures to Objects.
Addison-Wesley Professional, 8th edition.

📄 Jones, B. (2014).
Sams teach yourself C++ in one hour a day.
Sams.