Asynchronous Programming with Futures, Future Threads, and Traps

By Dong Nguyen

Introduction

Often processes execute in parallel by enabling applications to be multi-threaded, allowing system resources of multi-processor machines to process more efficiently and reduce computing time. Sometimes it is necessary to use a thread to process information and return its result. This can be done by passing an object, which will hold the result, to the constructor of the thread during thread creation. However, there is a danger of using the result too early, when the thread has not yet completed. This article discusses different approaches to solving this problem by presenting iterations of a simple application using a single-threaded model; using a multi-threaded model; using conditions for synchronization; using futures and future threads for deferred blocking; and using traps to decouple completion times between threads. Finally, the advantages and disadvantages of each approach are outlined.

Note: In the examples, the classes prefixed with `os_` are from Recursion Software's C++ libraries. The `os_stopwatch` class is used to measure the amount of elapsed time. The `os_thread` class is a façade for the low-level threading primitive of the operating system. The `os_condition` and `os_condition_mutex` classes are also façades of the their corresponding threading primitives. The `os_future, os_future_thread`, and `os_trap` classes support deferred blocking functionality.

Single vs. Multi Threaded

The following code simulates data processing in single-threaded operation:

Figure 1. First iteration with single-thread

```
#include <ospace/std/iostream>
#include <ospace/std/utility>
#include <ospace/time.h>

const int RESULT_SIZE = 5;


// initialization for result holder
int result[ RESULT_SIZE ];

void init_results()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    result[ i ] = -1;
    }
  }
```

```
void*
work( void* args )
  {
  pair< int, int* > argument = *(pair< int, int* >*)args;
  int index = argument.first;

  // amount of work will vary based on value of index
  for( int i = 0; i < index; i++ )
    for( int j = 0; j < 1000000; j++ )
      100000 / 50;

  cout << index << " " << *argument.second << endl;
  // set result to a valid value
  *argument.second = index;
  cout << index << " " << *argument.second << endl;

  return 0;
  }


void work()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    work( (void*)new pair< int, int* >( i, &result[ i ] ) );
    }
  }


void consume()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    cout << "main thread: " << i << " " << result[ i ] << endl;
    }
  }


int
main()
  {
  os_time_toolkit init_time;

  init_results();

  os_stopwatch stopwatch;
  stopwatch.start();

  // perform work and ready the results
  work();

  // consume results
  consume();

  stopwatch.stop();
  cout << "stopwatch.lap() = " << stopwatch.lap() << endl;
  return 0;
  }
```

Here is the same code with modifications for multi-threaded operation:

Figure 2. Second iteration with multiple threads

```
#include <ospace/std/iostream>
#include <ospace/std/utility>
#include <ospace/thread.h>
#include <ospace/time.h>

const int RESULT_SIZE = 5;


// initialization for result holder
int result[ RESULT_SIZE ];

void init_results()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    result[ i ] = -1;
    }
  }


os_barrier barrier( RESULT_SIZE + 1 );

void*
work( void* args )
  {
  pair< int, int* > argument = *(pair< int, int* >*)args;
  int index = argument.first;

  // amount of work will vary based on value of index
  for( int i = 0; i < index; i++ )
    for( int j = 0; j < 1000000; j++ )
      100000 / 50;

  cout << index << " " << *argument.second << endl;
  // set result to a valid value
  *argument.second = index;
  cout << index << " " << *argument.second << endl;

  barrier.wait();
  return 0;
  }


// initialization of worker threads
void init_workers()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    // pass variable to hold result to the worker thread
    os_thread::create_thread( work, (void*)new pair< int, int* >( i, &result[ i ] ) );
    }
  }


void consume()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    cout << "main thread: " << i << " " << result[ i ] << endl;
    }
  }


int
main()
```

```
  {
  os_thread_toolkit init_thread;
  os_time_toolkit init_time;

  init_results();

  os_stopwatch stopwatch;
  stopwatch.start();

  // create worker threads to perform work and ready the results
  init_workers();

  // simulate work by main thread
  os_this_thread::sleep( 2 );

  // consume results
  consume();

  // allow all worker threads to finish before proceeding
  barrier.wait();

  stopwatch.stop();
  cout << "stopwatch.lap() = " << stopwatch.lap() << endl;
  return 0;
  }
```

In Figure 1, the application performs as expected. The code calls the function `void* work( void* args )` and readies the results in the integer array `result.` Unfortunately, the code is less than optimal, since the results from the work done are independent from one another and not executed in parallel. The impact is code running slower than it should on multi-processor machines. In Figure 2, the application utilizes threads, enabling parallel processing. The changes include (1) spawning of threads (passing the location to store result) by calling the function `void init_workers()` and (2) modifying the function `void* work( void* args )`to use the argument-result pair argument. The execution time of the multi-threaded code now runs shorter than previously. However, the main thread consumes the results too early, which can be devastating to the stability and/or validity of the application.

Conditions

One possible remedy to this negative side effect is to use condition variables to synchronize the consumption of the results. A condition is a synchronization object that enables a thread to test an arbitrary condition (represented by the condition variable) and block until the state of the condition changes, all under the protection of a mutex. Here is an example of code with condition variables, conditions, and mutexes in place:

Figure 3. Third iteration with conditions and mutexes

```
#include <ospace/std/iostream>
#include <ospace/std/utility>
#include <ospace/thread.h>
#include <ospace/time.h>

const int RESULT_SIZE = 5;


// initialization for result holder
int result[ RESULT_SIZE ];

void init_results()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    result[ i ] = -1;
    }
  }
```

```
// initialization for synchronization mechanism
os_condition_mutex* result_mutex[ RESULT_SIZE ];
os_condition* result_cond[ RESULT_SIZE ];

void init_conds()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    result_mutex[ i ] = new os_condition_mutex;
    result_cond[ i ] = new os_condition( *result_mutex[ i ] );
    }
  }

void fin_conds()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    delete result_cond[ i ];
    delete result_mutex[ i ];
    }
  }


void*
work( void* args )
  {
  pair< int, int* > argument = *(pair< int, int* >*)args;
  int index = argument.first;
  try
    {
    result_mutex[ index ]->lock();

    // amount of work will vary based on value of index
    for( int i = 0; i < index; i++ )
      for( int j = 0; j < 1000000; j++ )
        100000 / 50;

    cout << index << " " << *argument.second << endl;
    // set result to a valid value
    *argument.second = index;
    cout << index << " " << *argument.second << endl;

    result_cond[ index ]->signal();
    result_mutex[ index ]->unlock();
    }
  catch( os_thread_toolkit_error& ex )
    {
    cout << ex.what() << endl;
    }
  return 0;
  }


// initialization of worker threads
void init_workers()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    // pass variable to hold result to the worker thread
    os_thread::create_thread( work, (void*)new pair< int, int* >( i, &result[ i ] ) );
    }
  }


void consume()
```

```
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    try
      {
      result_mutex[ i ]->lock();
      // Loop and check the condition. There could be spurious wakeups.
      while( result[ i ] == -1 )
        result_cond[ i ]->wait();
      result_mutex[ i ]->unlock();

      cout << "main thread: " << i << " " << result[ i ] << endl;
      }
    catch( os_thread_toolkit_error& ex )
      {
      cout << ex.what() << endl;
      }
    }
  }


int
main()
  {
  os_thread_toolkit init_thread;
  os_time_toolkit init_time;

  init_results();
  init_conds();

  os_stopwatch stopwatch;
  stopwatch.start();

  // create worker threads to perform work and ready the results
  init_workers();

  // consume results
  consume();

  stopwatch.stop();
  cout << "stopwatch.lap() = " << stopwatch.lap() << endl;

  fin_conds();
  return 0;
  }
```

In Figure 3, the changes involve (1) the creation of the set of condition-mutex synchronization objects by calling the function `void init_conds()`, (2) modifying the function `void* work( void* args )` to signal the main thread of the result notification by invoking the method `os_condition::signal()`, and (3) modifying the function `void consume()`to block access to the result until it is available by invoking the method `os_condition::wait()`. The conditions force the main thread to delay consumption of the result until the worker threads have assigned the correct, or valid, value to the result. The application now behaves as expected, although this method of synchronization can be laborious since the user has to manually keep track of the corresponding condition variables, conditions, and mutexes. In this case, matching the three objects is trivial given that the objects are each in their own array and related to one another by a common index. But imagine the complexity if the results were not so interrelated and the number of results were larger. Managing the synchronization would be a huge undertaking, which is error-prone and could lead to hard-to-debug code.

Futures and Future Threads

This leads us to futures and future threads. A future holds the result of the user-supplied function given to the thread and blocks if the result is not yet ready. A future thread can be used to process information and store the value in a future for later use, when it is done processing. Figure 4 illustrates the usage of the future and future thread classes.

Figure 4. Fourth iteration with futures and future threads

```
#include <ospace/std/iostream>
#include <ospace/thread.h>
#include <ospace/time.h>

const int RESULT_SIZE = 5;


void*
work( void* args )
  {
  int index = *(int*)args;
  try
    {
    // amount of work will vary based on value of index
    for( int i = 0; i < index; i++ )
      for( int j = 0; j < 1000000; j++ )
        100000 / 50;

    cout << index << " " << "no need for result holder anymore" << endl;
    }
  catch( os_thread_toolkit_error& ex )
    {
    cout << ex.what() << endl;
    }
  return (int*)index;
  }


// initialization of worker threads
os_future_thread* future_thread[ RESULT_SIZE ];

void init_workers()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    future_thread[ i ] = new
      os_future_thread
        (
        (os_future_thread_func_t)work,
        (void*)new int( i )
        );
    future_thread[ i ]->start();
    }
  }

void fin_workers()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    delete future_thread[ i ];
    }
  }


void consume()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
```

```
    try
      {
      os_future future_result = future_thread[ i ]->result();
      int result = (int)future_result.redeem();
      cout << "main thread: " << i << " " << result << endl;
      }
    catch( os_thread_toolkit_error& ex )
      {
      cout << ex.what() << endl;
      }
    }
  }


int
main()
  {
  os_thread_toolkit init_thread;
  os_time_toolkit init_time;

  os_stopwatch stopwatch;
  stopwatch.start();

  // create worker threads to perform work and ready the results
  init_workers();

  // consume results
  consume();

  stopwatch.stop();
  cout << "stopwatch.lap() = " << stopwatch.lap() << endl;

  fin_workers();
  return 0;
  }
```

In Figure 4, the changes involve (1) replacing the spawning of threads to spawning of future threads (no longer passing the location to store the results) in the function `void init_conds()`, (2) removing the usage of the result storage and condition-mutex synchronization mechanisms from the `function void* work( void* args )`, (3) modifying the function `void* work( void* args )` to return the results, (4) removing the usage of the result storage and condition-mutex synchronization mechanisms from the function `void consume()`, (5) modifying the function `void consume()to` use the future object to retrieve the results by invoking the method `os_future::redeem()`, and (6) removing the set of condition-mutex synchronization objects and the storage for the results. The potential to use the result too early is eliminated. The responsibility to synchronize access to the shared resource is placed on the future and future thread objects resulting in less user prone errors.

Traps

In addition to futures and future threads, the code can be further optimized if the order of consuming the results is not a concern. Traps are designed specifically to encapsulate this concept. A trap is a container for futures and returns the next future that is available with a result. With minimal code change, utilization of traps minimizes the wait time between result consumption, as illustrated below.

Figure 5. Fifth iteration with traps

```
#include <ospace/std/iostream>
#include <ospace/thread.h>
#include <ospace/time.h>

const int RESULT_SIZE = 5;
```

```
void*
work( void* args )
  {
  int index = *(int*)args;
  try
    {
    // amount of work will vary based on value of index
    for( int i = 0; i < index; i++ )
      for( int j = 0; j < 1000000; j++ )
        100000 / 50;

    cout << index << " " << "no need for result holder anymore" << endl;
    }
  catch( os_thread_toolkit_error& ex )
    {
    cout << ex.what() << endl;
    }
  return (int*)index;
  }


// initialization of worker threads
os_future_thread* future_thread[ RESULT_SIZE ];
os_trap trap;

void init_workers()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    future_thread[ i ] = new
      os_future_thread
        (
        (os_future_thread_func_t)work,
        (void*)new int( i )
        );
    future_thread[ i ]->start();
    trap.add_to_trap( future_thread[ i ]->result() );
    }
  }

void fin_workers()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    delete future_thread[ i ];
    }
  }


void consume()
  {
  for( int i = 0; i < RESULT_SIZE; i++ )
    {
    try
      {
      os_future future_result = trap.get_next();
      int result = (int)future_result.redeem();
      cout << "main thread: " << i << " " << result << endl;
      }
    catch( os_thread_toolkit_error& ex )
      {
      cout << ex.what() << endl;
      }
    }
  }
```

```
int
main()
  {
  os_thread_toolkit init_thread;
  os_time_toolkit init_time;

  os_stopwatch stopwatch;
  stopwatch.start();

  // create worker threads to perform work and ready the results
  init_workers();

  // consume results
  consume();

  stopwatch.stop();
  cout << "stopwatch.lap() = " << stopwatch.lap() << endl;

  fin_workers();
  return 0;
  }
```

In Figure 5, the changes involve adding threads to the trap during future thread spawning and modifying the `consume()` function to retrieve the next available result indicated by the trap. Traps are advantageous in scenarios where the thread completion times vary widely.

Conclusion

In summary, single-threaded applications do not efficiently use the resources provided by multi-processor machines; however, just adding thread capabilities could destabilize and/or invalidate the application. Adding conditions and mutexes helps, but they add more complexity to the code. To address these problems, futures can be use to store the result for later use and block if the result has not been received. Futures do not necessarily eliminate wait time; they merely minimize it. Also, concepts such as future threads and traps can help to utilize futures more effectively. Future threads allow the application to start their processing at any arbitrary point and return the result, encapsulated in a future, anytime after it is ready. Traps are containers for futures, which return the next future that is available with a result. This frees the application from being dependent on the order in which the future threads will finish processing.

For additional technical information on Recursion Software
products and programs or for information on how to order and evaluate
Recursion Software technology, contact us today!

Recursion Software, Inc.
2591 North Dallas Parkway, Suite 200
Frisco, Texas 75034
1.800.727.8674 or 972.731.8800
www.recursionsw.com