Last Time: Multithreading

- Covered C++ threads, mutexes
- Threads are kind of finicky...
  - If a thread function throws an exception, the entire program is terminated
  - If a `std::thread` object’s destructor is called when it is still joinable, the entire program is terminated

- C++ has several other threading abstractions to simplify multithreaded programming
Asynchronous Function Invocations

- Can also execute a function **asynchronously**
  
  ```
  #include <future>
  // calls fn(a, b, c) asynchronously;
  // returns a future to access the result
  future f = async(fn, a, b, c);
  ```
  
  - async() call returns immediately

- **future** has a number of member functions:
  
  - **get()** returns the result of fn(a, b, c), blocking if invocation is not yet completed
  - **wait()** blocks until the invocation is completed, but doesn’t return the function’s result
  - **valid()** returns true if invocation is completed, false otherwise
Asynchronous Function Invocations (2)

- What if \( fn(a, b, c) \) throws an exception?
  
  ```java
  future f = async(fn, a, b, c);
  int result = f.get();
  ```

- When `get()` is called, it will throw any exception that was thrown by the invoked function.
  
  - The future acts exactly like the function would have acted, but invocation took place in background.

- Does this mechanism actually require that `fn` execute on a background thread?
  
  - **No!** The `future` could execute `fn(a, b, c)` when `get()` is called...
Asynchronous Function Invocations (3)

- The `async()` function can choose from several launch policies when executing a function:
  
  \[ f = \text{async}(\text{launch::async, fn, a, b, c}) \]
  
  - Runs \( \text{fn}(a, b, c) \) on a background thread

  \[ f = \text{async}(\text{launch::deferred, fn, a, b, c}) \]
  
  - Runs \( \text{fn}(a, b, c) \) when \( f.\text{get()} \) or \( f.\text{wait()} \) is called (no background thread is involved!)

  \[ f = \text{async}(\text{launch::async | launch::deferred, fn, a, b, c}) \]
  
  - Lets the C++ runtime choose whether to use a thread, or just defer invocation of the function

  **This is the default behavior, if unspecified!**
Asynchronous Function Invocations (4)

- `async()` function and `future` objects provide a **task-based approach** to multithreading, instead of a thread-based approach.
- If you simply want to execute slow tasks in the background, use the task-based approach.
  - Behaves better with functions that throw.
  - Cleans up thread objects automatically.
- Currently, `async()` doesn’t scale very well.
  - Many C++ standard library implementations don’t use a thread-pool to run asynchronous tasks.
  - Better off using a real thread-pool if you have many tasks to execute in background.
Asynchronous Function Invocations (5)

- If you need more sophisticated multithreading or thread-synchronization, just use threads
  - Parallel computing, high-throughput multithreaded data processing, etc.
  - Sophisticated thread-synchronization primitives like barriers, etc.
Last time, gave an example of a thread-safe FIFO

template<typename T, int maxElems>
class ThreadSafeFIFO {

typedef struct list_elem {
    T *data;
    list_elem *next;
};

mutex m; // To guard FIFO internals
int numElems; // How many elems in FIFO?
list_elem *head, *tail; // Linked list of elems

public:
...
};
Want a `take()` member function on our FIFO

Wait until an element becomes available

A naïve approach:

```cpp
T * take() {
    T *result = nullptr;
    while (true) {
        lock_guard<mutex> lock(m);
        if (numElems > 0) {
            result = ...; // Remove first element
            break;
        }
    }
    return result;
}
```
A naïve approach:

```cpp
T * take() {
    T *result = nullptr;
    while (true) {
        lock_guard<mutex> lock(m);
        if (numElems > 0) {
            result = ...; // Remove first element
            break;
        }
    }
    return result;
}
```

Problem: This code actively waits for a value

Will use the CPU, keeping other threads from running
Passive Waiting

- Should almost **never** require active polling like this!
  - A rare exception is when interacting directly with hardware e.g. in an embedded system
- Strongly prefer **passive waiting**: the thread is suspended while waiting for a value
- The waiting thread doesn’t know when a new value will show up...
  - Another thread will call `put()`, which will allow a waiting thread to proceed
  - Need a way for one thread to notify the other, when it is capable of proceeding
- Can implement this with **condition variables**
Passive Waiting (2)

- Condition variables allow threads to be notified when some condition becomes true
  - The program implements the condition
  - The condition variable is simply a synchronization primitive
- Condition variables must be used in the context of a mutex
  - The two communicating threads are manipulating shared state...
  - The mutex protects that shared state
Waiting Threads

- Need to have a mutex and a condition variable to use:
  ```
  #include <mutex>
  #include <condition_variable>
  mutex m;
  condition_variable cv;
  ```
- Waiting thread can wait on the condition variable, after it acquires a lock on the mutex:
  ```
  unique_lock lock(m);
  cv.wait(lock); // Suspends the calling thread
  ```
- Must use a `unique_lock` so that the condition variable can manipulate the mutex:
  ```
  Mutex is unlocked right before the thread suspends, so that other threads can access the shared state
  ```
  ```
  Mutex will be re-locked before cv.wait(lock) returns
  ```
Notifying Threads

- Need to have a mutex and a condition variable to use:
  
  ```
  #include <mutex>
  #include <condition_variable>
  mutex m;
  condition_variable cv;
  ```

- A notifying thread can use the condition variable to notify one waiting thread, or all waiting threads
  
  ```
  cv.notify_one();
  cv.notify_all();
  ```

  - If no threads are waiting, these are no-ops

- Note: no mutex is required when notifying
  
  - Usually, the notifier also holds a mutex when notifying

- Only call `notify_all()` when it makes sense to wake up multiple threads! Otherwise it's just inefficient.
Add a condition variable to our thread-safe FIFO

```cpp
template<typename T, int maxElems>
class ThreadSafeFIFO {
  typedef struct list_elem {
    T *data;
    list_elem *next;
  };
  mutex m;           // To guard FIFO internals
  condition_variable cv;   // For passive waiting
  int numElems;      // How many elems in FIFO?
  list_elem *head, *tail;  // Linked list of elems

public:
  ...;
};
```
Update our `take()` function to passively wait:

```cpp
T * take() {
    T *result = nullptr;
    unique_lock lock(m);
    while (numElems == 0) // Wait for some data
        cv.wait(lock);
    assert(numElems > 0);
    result = ...; // Remove first element
    return result;
}
```

Much simpler code!

- Recall: mutex is unlocked before suspend, and then re-locked after notification
Why do we wait in a loop?!

```cpp
T * take() {
    T *result = nullptr;
    unique_lock lock(m);
    while (numElems == 0) // Wait for some data
        cv.wait(lock);
    assert(numElems > 0);
    result = ...; // Remove first element
    return result;
}
```

Some condition-variable implementations may spuriously wake up waiting threads (?!)

Always wait in a loop, until the required condition actually becomes true.
Also need to update `put()` to notify any waiters

```cpp
void put(T *elem) {
    // Don’t need unique_lock here
    lock_guard lock(m);
    ... // Put element onto back of FIFO
    cv.notify_one();
}
```

- Only notify one waiter – there is only one new value

- **Note:** when `notify_one()` is called, the mutex is still held by this thread
  - When this thread releases lock, some other thread will acquire the lock and will be able to proceed
Our thread-safe FIFO has a maximum size...

```cpp
template<typename T, int maxElems>
class ThreadSafeFIFO {
...
```

Want `put()`ters to also passively wait if the FIFO doesn't have room for more elements

- Use two condition variables – one for putters, and the other for takers
  ```cpp
  condition_variable put_cv; // For putters
  condition_variable take_cv; // For takers
  ...
  ```
Update the `put()` member function:

```cpp
void put(T *elem) {
    unique_lock lock(m);
    while (numElems == maxElems)
        put_cv.wait(lock);
    assert(numElems < maxElems);
    ... // Put element onto back of FIFO
    take_cv.notify_one();
}
```

Also update `take()` to call `put_cv.notify_one()` after removing a value from the FIFO.
Will shoot for one more assignment in CS11 Advanced C++
- Animate non-player characters
- Allow the player to move around
- Create a basic game loop