Lecture 2 – Tasks, Threads and Synchronization (in Modern C++)

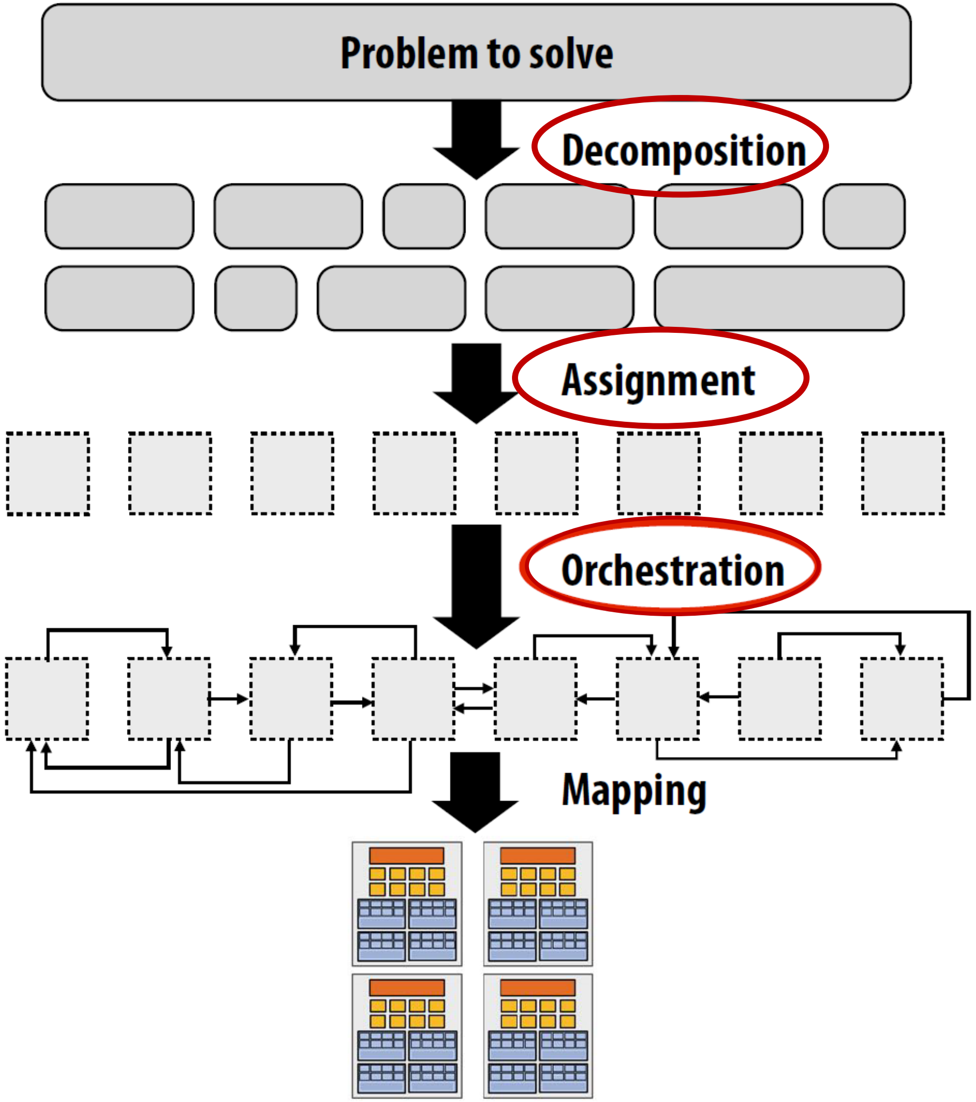
CS3211 Parallel and Concurrent Programming

Outline

- Tasks
- A bit of C++ history
- Ownership, lifetime, and RAII in C++
- Threads in C++
- Synchronizing threads in C++
  - Shared data: mutex
  - Concurrent actions: condition variables

Program Execution

- 3 main steps:
  - Decomposition of the computations
  - Assigning tasks to threads
  - Orchestration
    - Structuring communication
    - Adding synchronization to preserve dependencie
    - Organizing data structures in memory
    - Scheduling tasks
  - Mapping of threads to physical cores

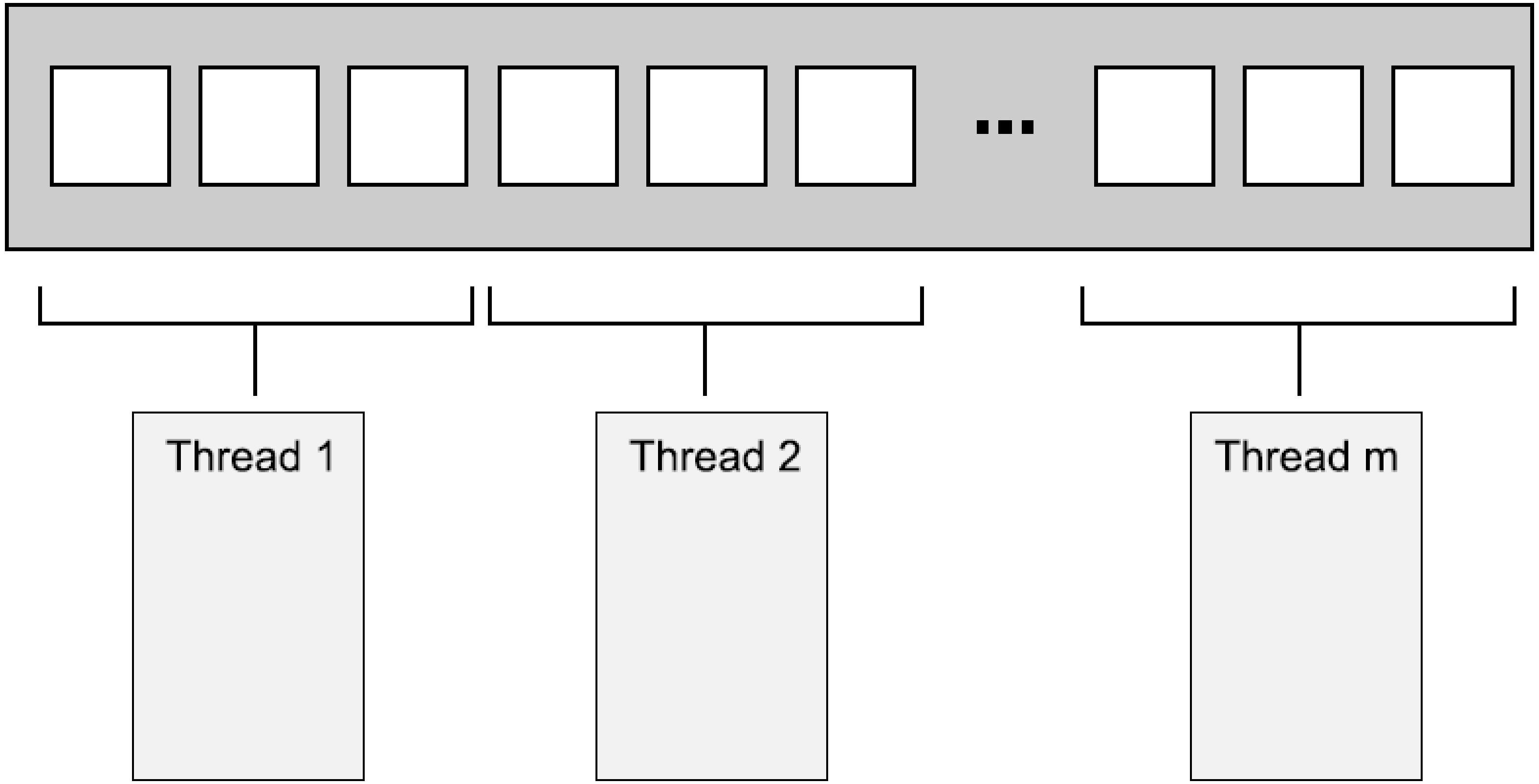


Distributing work to threads – **task** parallelism

- Divide the work into tasks to make the threads specialists
  - Same types of tasks assigned to the same thread (aka pipeline)
  - Divide the work by task type to separate concerns
- Dividing a sequence of tasks among threads to achieve a complex solution
  - Pipeline: each thread is responsible for a stage of a pipeline
- Use task pools and the number of threads to serve the task pool

Distributing work to threads – **data** parallelism

- Dividing data



Outline

- Tasks
- A bit of C++ history
- Ownership, lifetime, and RAII in C++
- Threads in C++
- Synchronizing threads in C++
  - Shared data: mutex
  - Concurrent actions: condition variables

C++ history

- 1998 –the original C++ standard published
  - No support for multithreading!
  - Use external libraries to manage threads in your C/C++ programs
    - Pthread
- 2011 – C++11 (or C++0x) standard published
  - new “train model” of releases
  - support of multithreaded programs
- 2014
- 2017
- 2020
- 2023

C++98

- Does not acknowledge the existence of threads
- Effects of language elements assume a sequential abstract machine
  - No memory model
- Multithreading was dependent on compiler-specific extensions
  - C APIs, such as POSIX C standard and Microsoft Windows API, used in C++
  - Very few formal multithreading-aware memory models provided by compiler vendors
  - Application frameworks, such as Boost and ACE, wrap the underlying platform-specific APIs
    - Provide higher-level facilities for multithreading

C++11 multithreading

- Write portable multithreaded code with guaranteed behavior
  - Multithreading without relying on platform-specific extensions
- Thread-aware memory model
- Includes classes for managing threads, protecting shared data, synchronization between threads, low-level atomic operations
- Use of concurrency to improve application performance
  - Take advantage of the increased computing power
  - Low abstraction penalty - C++ classes wrap low-level facilities
  - Low-level facilities: atomic operations library

Managing threads

- Every program has at least one thread
  - Started by the C++ runtime
  - Runs `main()`
- Use `std::thread` to add threads

Creating a thread

- Identifying a thread using `get_id()`

```
#include <iostream>
#include <thread>

void hello()
{
    std::cout<<"Hello Concurrent World\n";
}

int main()
{
    std::thread t(hello);
    t.join();
}

```

Starting threads

- There are different ways to start threads:
  - Using a function (like pthreads)
  - Lines 1-2: Thread with a function

```cpp
1  void do_some_work();
2  std::thread my_thread(do_some_work);
```

  - Using a function object
  - Lines 6-16: Thread with a function object (callable type)

```cpp
6   class background_task
7   {
8   public:
9       void operator()() const
10      {
11          do_something();
12          do_something_else();
13      }
14  };
15  background_task f;
16  std::thread my_thread(f);
```

Starting threads (cont)

Line 20 declares a function that returns a thread:

- Declares a `my_thread` function that takes a single parameter (of type `pointer-to-a-function-taking-no-parameters-and-returning-a-background_task-object`) and returns a `std::thread` object
- Does not launch a new thread!

```
20 std::thread my_thread(background_task());
```

Starting threads (cont)

Line 20: Declare a function that returns a thread

```
20  std::thread my_thread(background_task());
```

Lines 24-25: Thread with a function object

```
24  std::thread my_thread((background_task()));
25  std::thread my_thread{background_task()};
```

Lines 30-33: Thread with a lambda expression (local function instead of a callable object)

```
30  std::thread my_thread([]{
31      do_something();
32      do_something_else();
33  });
```

Starting threads - summary

```
1   void do_some_work();
2   std::thread my_thread(do_some_work);
```

4 different ways:

1-2 Thread with a function

```
6   class background_task
7   {
8   public:
9       void operator()() const
10      {
11          do_something();
12          do_something_else();
13      }
14  };
15  background_task f;
16  std::thread my_thread(f);
```

6-16 Thread with a function object (callable type)

```
20  std::thread my_thread(background_task());
```

```
24  std::thread my_thread((background_task()));
25  std::thread my_thread{background_task()};
```

24-25 Thread with a function object

```
30  std::thread my_thread([]{
31      do_something();
32      do_something_else();
33  });
```

30-33 Thread with a lambda expression (local function instead of a callable object)

Wait or detach?

- Wait for a thread to finish
  - Use `join()` on the thread instance, exactly once
  - Use `joinable()` to check
- Detach the thread
  - Use `detach()`
  - Extra care is needed with local variables passed to the thread

Waiting

- Make sure to join the thread even when there is an exception!

```cpp
void f()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread t(my_func);
    try
    {
        do_something_in_current_thread();
    }
    catch(...)
    {
        t.join();
        throw;
    }
    t.join();
}
```

```cpp
#include <thread>

void do_something(int& i)
{
    ++i;
}

struct func
{
    int& i;

    func(int& i_):i(i_){}

    void operator()()
    {
        for(unsigned j=0;j<1000000;++j)
        {
            do_something(i);
        }
    }
};

void do_something_in_current_thread()
{}

void f()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread t(my_func);
    try
    {
        do_something_in_current_thread();
    }
    catch(...)
    {
        t.join();
        throw;
    }
    t.join();
}

int main()
{
    f();
}
```

Detach

- Local variables passed as parameters to the thread function might end their lifetime before the thread ends

```cpp
#include <thread>

void do_something(int& i)
{
    ++i;
}

struct func
{
    int& i;

    func(int& i_):i(i_){}

    void operator()()
    {
        for(unsigned j=0;j<1000000;++j)
        {
            do_something(i);
        }
    }
};


void oops()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread my_thread(my_func);
    my_thread.detach();
}

int main()
{
    oops();
}
```

Passing arguments to a thread function

- Lines 1-2: Passing a function and arguments by value

```
1  void f(int i,std::string const& s);
2  std::thread t(f,3,"hello");
```

- Lines 11-18: oops might exit before the `buffer` is converted to `std::string` within the new thread (passing a reference to buffer)

```
11 void f(int i,std::string const& s);
12 void oops(int some_param)
13 {
14     char buffer[1024];
15     sprintf(buffer, "%i",some_param);
16     std::thread t(f,3,buffer);
17     t.detach();
18 }
```

- Lines 21-28: the conversion happens before passing the argument to the thread through an explicit cast

```
21 void f(int i,std::string const& s);
22 void not_oops(int some_param)
23 {
24     char buffer[1024];
25     sprintf(buffer,"%i",some_param);
26     std::thread t(f,3,std::string(buffer));
27     t.detach();
28 }
```

Passing arguments by reference

- Lines 31-39: data is passed by value (copy)

```
31 void update_data_for_widget(widget_id w,widget_data& data);
32 void oops_again(widget_id w)
33 {
34     widget_data data;
35     std::thread t(update_data_for_widget,w,data);
36     display_status();
37     t.join();
38     process_widget_data(data);
39 }
```

- Line 41: Wrap the arguments in std::ref

```
41 std::thread t(update_data_for_widget,w,std::ref(data));
```

Ownership in C++

- An owner is an object containing a pointer to an object allocated by `new` for which a `delete` is required
- Every object on the free store (heap, dynamic store) must have exactly one owner.
- C++'s model of resource management is based on the use of constructors and destructors
  - For scoped objects, destruction is implicit at scope exit
  - For objects placed in the free store (heap, dynamic memory) using `new`, `delete` is required

❖ Objects can also be allocated using `malloc()` and deallocated using `free()` (or similar functions), but the techniques described for `new` and `delete` apply to those also

❖ Source: https://www.stroustrup.com/resource-model.pdf

RAII - Resource Acquisition Is Initialization

- C++ programming technique
- Binds the life cycle of a resource that must be acquired before use (allocated heap memory, thread of execution, open socket, open file, locked mutex, disk space, database connection—anything that exists in limited supply) to the lifetime of an object.

Lifetime

- The lifetime of an object begins when:
  - storage with the proper alignment and size for its type is obtained, and
  - its initialization (if any) is complete (including default initialization via no constructor or trivial default constructor) (except that of union and allocations by std::allocator::allocate)
- The lifetime of an object ends when:
  - if it is of a non-class type, the object is destroyed (maybe via a pseudo-destructor call), or
  - if it is of a class type, the destructor call starts, or
  - the storage which the object occupies is released, or is reused by an object that is not nested within it.
- Lifetime of an object is equal to or is nested within the lifetime of its storage, see storage duration.
- The lifetime of a reference begins when its initialization is complete and ends as if it were a scalar object.

  Note: the lifetime of the referred object may end before the end of the lifetime of the reference, which makes dangling references possible.

Ownership of a `thread`

- `std::thread` instances own a resource
  - Manage a thread of execution
- Instances of `std::thread` are
  - *Movable*
  - are not *copyable*
- (Same ownership semantics as `std::unique_ptr`)

```
void some_function();
void some_other_function();
std::thread t1(some_function);
std::thread t2=std::move(t1);
t1=std::thread(some_other_function);
std::thread t3;
t3=std::move(t2);
t1=std::move(t3);
```

Transferring ownership of a thread

- Lines 11-19: Transfer ownership out of function
- Lines 21-28: Transfer ownership into a function

```
11 std::thread f() {
12     void some_function();
13     return std::thread(some_function);
14 }
15 std::thread g() {
16     void some_other_function(int);
17     std::thread t(some_other_function,42);
18     return t;
19 }
```

```
21 void f(std::thread t);
22 void g()
23 {
24     void some_function();
25     f(std::thread(some_function));
26     std::thread t(some_function);
27     f(std::move(t));
28 }
```

Outline

- Tasks
- A bit of C++ history
- Ownership, lifetime and RAII in C++
- Threads in C++
- Synchronizing threads in C++
  - Shared data: mutex
  - Concurrent actions: condition variables

Synchronizing multiple threads

1. Concurrent access to shared data
   - Mutex
2. Concurrent actions
   - Condition variable
   - Monitor

Synchronizing concurrent accesses

- If all shared data is read-only – no problem
  - the data read by one thread is unaffected by another thread is reading the same data

- Modifying shared data comes with many challenges

Invariants

- Invariants – statements that are always true about a particular data structure
  - Often broken during an update on the data structure
  - Example: this variable `no_of_items` contains the number of items in the list
- Use invariants to reason about program correctness

Delete a node from a doubly linked list

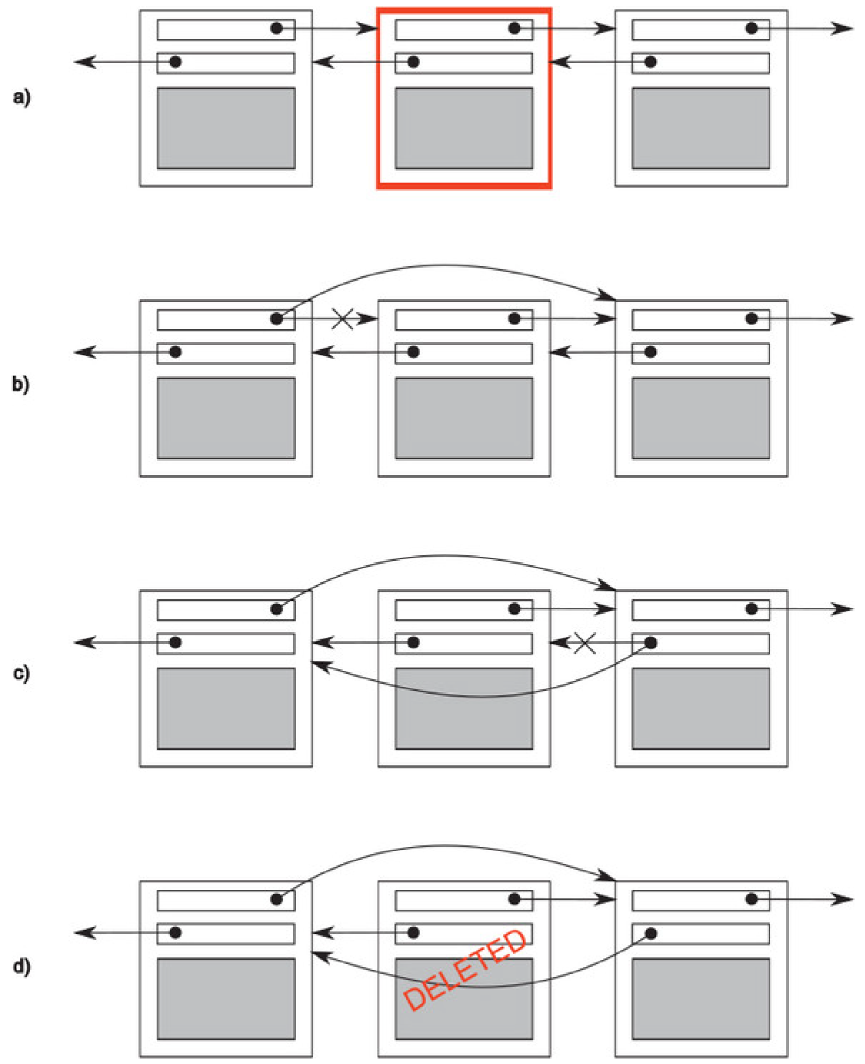
*Invariant is broken during the delete

a) Identify the node to delete: N.

b) Update the link from the node prior to N to point to the node after N.

c) Update the link from the node after N to point to the node prior to N.

d) Delete node N.



Problems with sharing data among threads

- Programs must be designed to ensure that changes to a chosen data structure are correctly synchronized among threads
  - Data structures are immutable (data never changes), or
  - Protect data using some locking: external or internal
- Invariants are often broken during an update on the data structure
  - Other threads might work with data while the invariant is broken
- Race conditions

Race conditions vs. data races

Race condition

- The outcome depends on the relative ordering of execution of operations on two or more threads
- The threads race to perform their respective operations
- Usually, *race condition* is a flaw that occurs when the timing or ordering of events affects a program’s correctness.

Data race

- A *data race* happens when there are two memory accesses in a program where both:
  - target the same location
  - are performed concurrently by two threads
  - are not reads
  - are not synchronization operations
- Causes undefined behavior

Avoiding race conditions

- Chances of the problematic execution sequence occurring increases
  - high load in the system
  - the operation is performed more time
- Simplest option: wrap your data structure with a protection mechanism
  - Ensure that only the thread performing a modification can see the intermediate states while the invariants are broken
  - C++ provides such mechanisms for locking

Synchronization primitive: mutex

- Mutex = *mut*ual *ex*clusion
- Provides *serialization*
  - Threads take turns accessing the data protected by the mutex

Mutex in C++

- Not recommended usage:
  1. Construct an instance of `std::mutex`,
  2. Lock it with a call to the `lock()` member function
  3. Unlock it with a call to the `unlock()` member function
  - Must remember to call `unlock()` on every code path out of a function, including those due to exceptions!
- Recommended usage: use `std::lock_guard` class template
  - Implements RAII idiom for a mutex
  - Locks the supplied mutex on construction and unlocks it on destruction

First example

- Global variable `some_list` is protected with a global instance of `std::mutex`

```cpp
1  #include <list>
2  #include <mutex>
3  #include <algorithm>
4  std::list<int> some_list;
5  std::mutex some_mutex;
6  void add_to_list(int new_value)
7  {
8      std::lock_guard<std::mutex> guard(some_mutex);
9      some_list.push_back(new_value);
10 }
11 bool list_contains(int value_to_find)
12 {
13     std::lock_guard<std::mutex> guard(some_mutex);
14     return std::find(some_list.begin(),some_list.end(),value_to_find)
15         != some_list.end();
16 }
```

- In C++17: template argument deduction enables omitting the template argument list (line 18 instead of lines 8 and 13)

```cpp
18  std::lock_guard guard(some_mutex);
```

What issue do you see with line 18?

Attempt on improving the usage of mutex

- Common to group the mutex and the protected data together in a class rather than use global variables
  - Encapsulate the functionality and enforce the protection

Issues when passing references

- The call to the user-supplied `func` function means that foo can pass in `malicious_function` to bypass the protection and then call `do_something()` without the mutex being locked

- Don't pass pointers and references to protected data outside the scope of the lock, whether by returning them from a function, storing them in externally visible memory, or passing them as arguments to user-supplied functions!

```cpp
class some_data
{
    int a;
    std::string b;
public:
    void do_something();
};
class data_wrapper
{
private:
    some_data data;
    std::mutex m;
public:
    template<typename Function>
    void process_data(Function func)
    {
        std::lock_guard<std::mutex> l(m);
        func(data);
    }
};
some_data* unprotected;
void malicious_function(some_data& protected_data)
{
    unprotected=&protected_data;
}
data_wrapper x;
void foo()
{
    x.process_data(malicious_function);
    unprotected->do_something();
}

```

Other types of locks

- `std::unique_lock` instance doesn't always own the mutex that it is associated with
  - Allows for locking the mutex later using `std::defer_lock`
- `std::lock()` function locks one or more mutexes at once without risk of deadlock
  - Use `std::adopt_lock` to indicate to `std::lock_guard` objects that the mutexes are already locked (should adopt the ownership of the existing lock on the mutex rather than attempt to lock the mutex in the constructor)
- `std::scoped_lock` instance accepts and locks a list of mutexes
  - Locks in the same way as `lock()`

Synchronizing multiple threads

1. Concurrent access to shared data
   - Mutex
2. Concurrent actions
   - Condition variable
   - Monitor – in tutorial 1

Waiting for an event or other condition

- One thread is waiting for a second thread to complete a task
  - Keep checking a flag in shared data (protected by a mutex) and have the second thread set the flag when it completes the task ~ wasteful
  - Waiting thread sleeps for short periods between the checks using the `std::this_thread::sleep_for()` function ~ less wasteful
  - Use condition variable to wait for an event to be triggered by another thread

```
bool flag;
std::mutex m;
void wait_for_flag()
{
    std::unique_lock<std::mutex> lk(m);
    while(!flag)
    {
        lk.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        lk.lock();
    }
}
```

Condition variable - definition

- A condition variable is associated with an event or other *condition*,

and

- One or more threads can *wait* for that condition to be satisfied
- How it works:
  - When the condition is satisfied, the thread can then *notify* one or more of the threads waiting on the condition variable
  - The notified threads wake up and continue processing

Two implementation in C++

`std::condition_variable`

- Works with `std::mutex`
- Simpler, lightweight, less overhead

`std::condition_variable_any`

- Works with anything mutex-like
- Potentially, additional costs in terms of size, performance, or OS resources

C++ Example

- 20: use a `std::unique_lock` rather than a `std::lock_guard`
- 21-22: If condition is satisfied, returns
- 21-22: If condition is NOT satisfied, `wait()` unlocks the mutex and puts the thread in a blocked or waiting state
  - until `notify_one()` is called (and must reacquire the lock before proceeding)

```
1  std::mutex mut;
2  std::queue<data_chunk> data_queue;
3  std::condition_variable data_cond;
4  void data_preparation_thread()
5  {
6      while(more_data_to_prepare())
7      {
8          data_chunk const data=prepare_data();
9          {
10             std::lock_guard<std::mutex> lk(mut);
11             data_queue.push(data);
12         }
13         data_cond.notify_one();
14     }
15 }
16 void data_processing_thread()
17 {
18     while(true)
19     {
20         std::unique_lock<std::mutex> lk(mut);
21         data_cond.wait(
22             lk,[]{return !data_queue.empty();});
23         data_chunk data=data_queue.front();
24         data_queue.pop();
25         lk.unlock();
26         process(data);
27         if(is_last_chunk(data))
28             break;
29     }
30 }
```

Condition variable behavior

- During a call to wait(), a condition variable
  - May check the supplied condition any number of times
    - Do not use a function with *side effects* for the condition check
  - Checks the condition with the mutex locked
  - Returns immediately if (and only if) the function provided to test the condition returns true
- *Spurious wake*
  - The waiting thread reacquires the mutex and checks the condition, but not in direct response from a notification

Cond var: an optimization over a busy-wait

- A basic, but inefficient, implementation

```
template<typename Predicate>
void minimal_wait(std::unique_lock<std::mutex>& lk,Predicate pred){
    while(!pred()){
        lk.unlock();
        lk.lock();
    }
}
```

- There is no guarantee about how the condition variable is implemented
  - Programmers must be prepared for both: spurious wakes and waking only when notified

Summary

- Introduced modern C++, std::thread and synchronization
  - Lifetime, ownership and RAII
- What's next?
  - Safely and efficiently using synchronization primitives to implement thread-safe data structures
  - Use lock-based primitives to enable synchronization at the right level of granularity
- Reference
  - C++ Concurrency in Action, Second Edition
    - Chapters 2, 3.1, 3.2, 4.1