Contemporary C++: Learning Modern C++ in a Modern Way

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 24/24

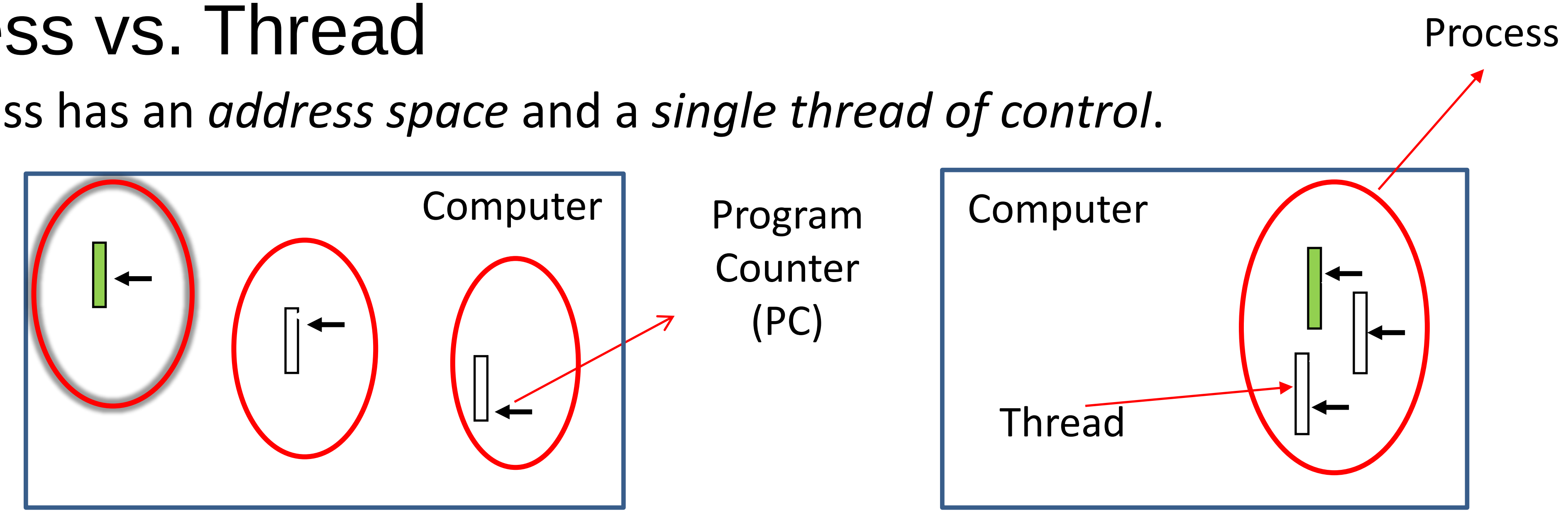
Session 24. More on Concurrency in C++: Multi-threaded programming in C++, Memory model, Synchronization and More

- C++ Threads: under the hood
- C++ Memory model
- Data races, critical sections, and mutual exclusion
- Thread safety
- C++ locks: lock_guard and unique_lock
- Condition variables
- Writing a few simple multi-threaded programs in C++
- Q&A

150 min (incl. Q & A)

Process vs. Thread

- Each process has an *address space* and a *single thread of control*.



Three processes with one thread

One process with three threads

- All threads have exactly the same address space, which means they also share same global variables → sharing global variables

- Processes
  - CPU sharing
  - Child process and child thread
  - Different states: *Running, Ready, Blocked*
- Different processes are independent.

- Threads
  - CPU sharing
  - Child thread
  - Different states: *Running, Ready, Blocked*
- Different threads aren’t independent.

- Thread is to process as process is to machine.

Threads … Operating System level definition

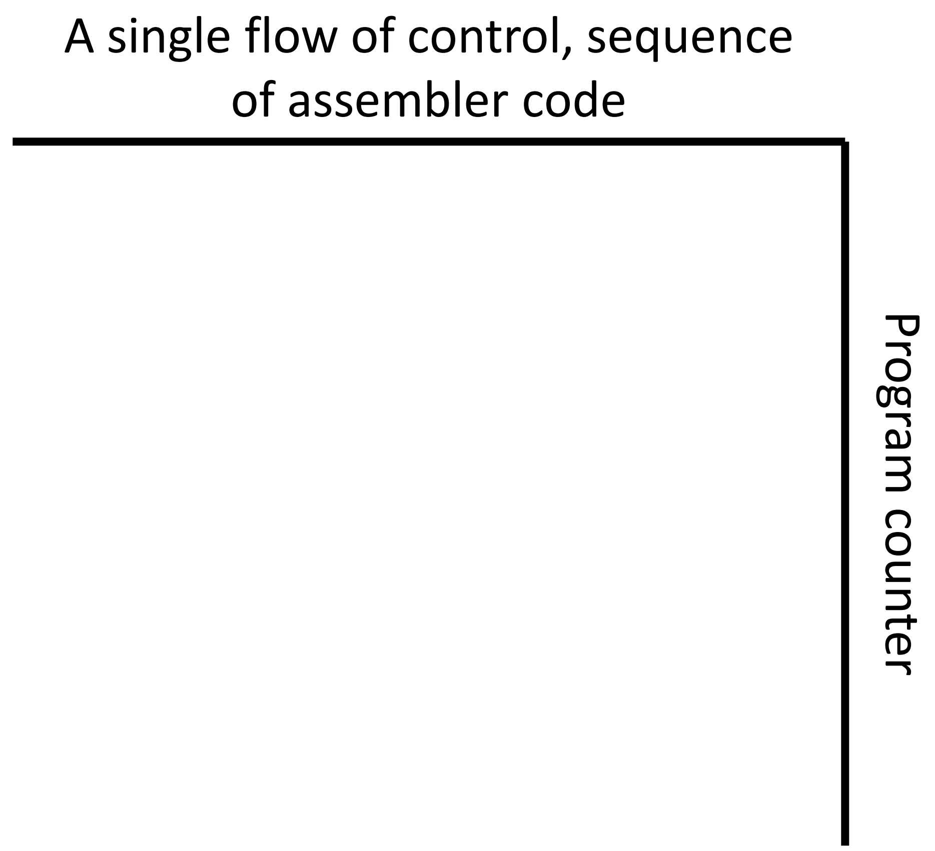
Threads … Operating System level definition

- Thread is an entity which exposes 4 properties

A single flow of control, sequence of assembler code

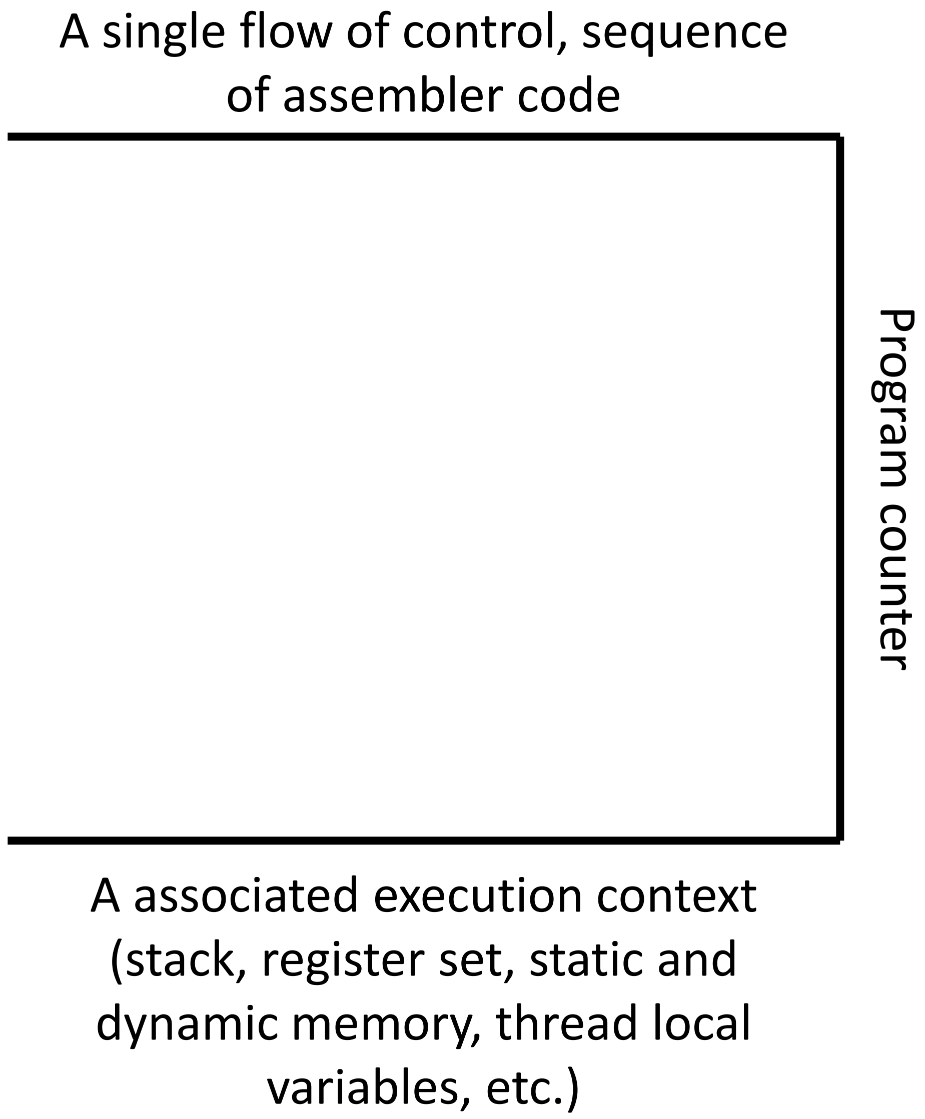
Threads … Operating System level definition

- Thread is an entity which exposes 4 properties



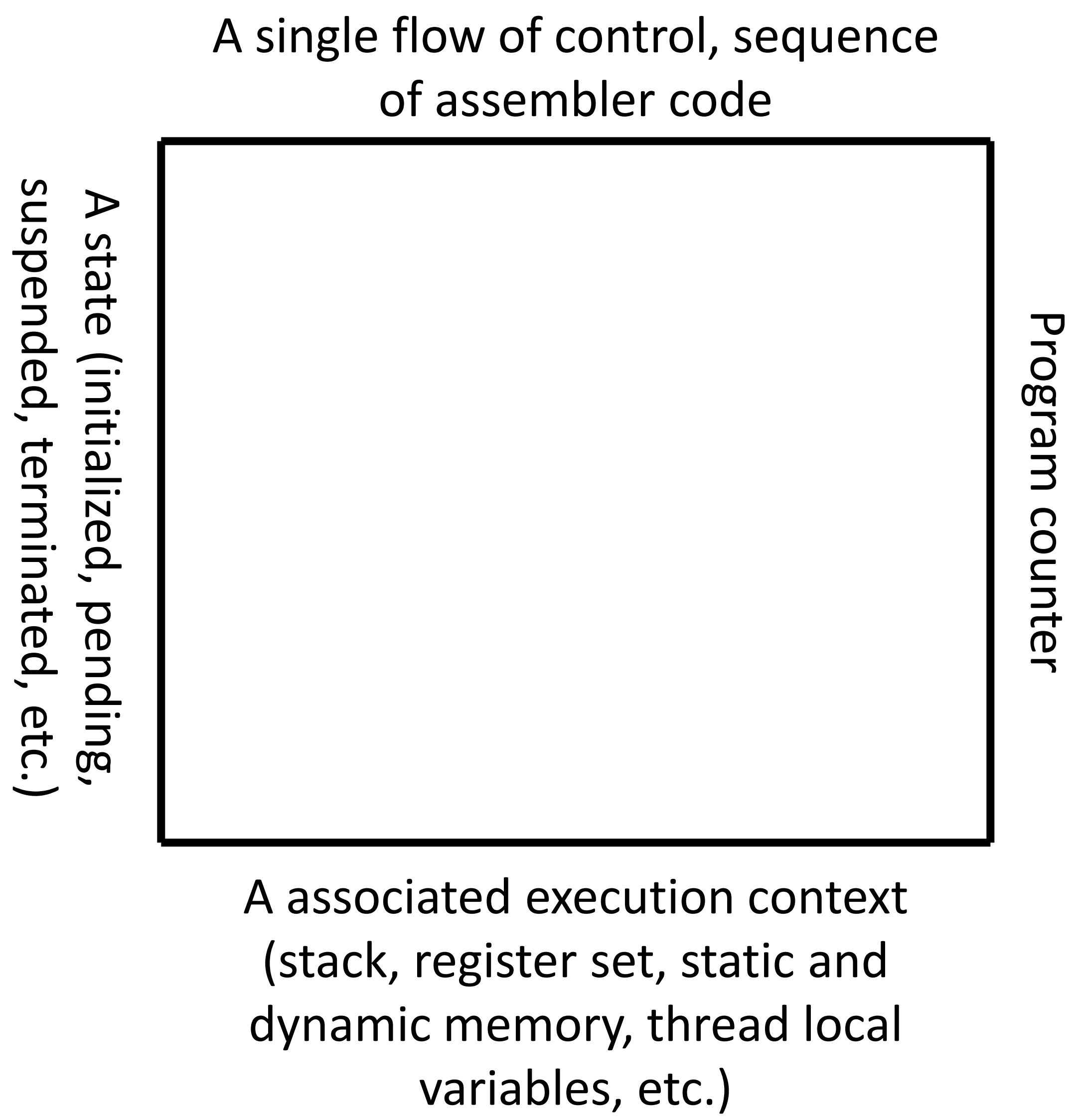
Threads … Operating System level definition

- Thread is an entity which exposes 4 properties


A single flow of control, sequence of assembler code
Program counter
A associated execution context (stack, register set, static and dynamic memory, thread local variables, etc.)

Threads … Operating System level definition

- Thread is an entity which exposes 4 properties


A single flow of control, sequence of assembler code
A state (initialized, pending, suspended, terminated, etc.)
Program counter
A associated execution context (stack, register set, static and dynamic memory, thread local variables, etc.)

Process vs. Thread cont.

Per thread items
Program counter
Call stack
Register set
Child threads
State

Per process items
Address space
Global variables
Open files
Child Process
…

ألفا
ALPHA

Threads … formal Definitions

Threads … formal Definitions

- A *thread of execution* (also known as a *thread*) is a single flow of control within a program.

C++ committee draft, section 6.8.2.p1 [intro.multithread] *page 66.*

Threads … formal Definitions

- A *thread* is the system-level representation of a task in a program.

Bjarne Stroustrup, 4th, page 115.

- A *thread* of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

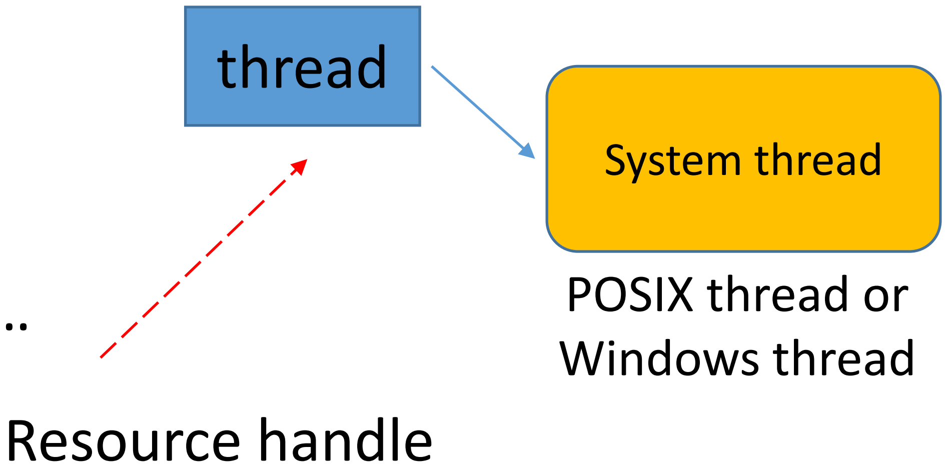
Wikipedia:
https://en.wikipedia.org/wiki/Thread_(computing)

Threads … informal Definitions

- A *lightweight* Process
- A *virtual* CPU

thread as a Handle

- A C++ thread represents a *system thread*.
- C++ thread = OS thread
- OS thread: Linux thread, Windows thread, MacOS thread, …
  - POSIX thread
- Heavyweight, pre-emptive, independent, …
- A Windows thread implementation



1
```
// Windows thread: VS 2017 implementation
typedef unsigned int _Thrd_id_t;
typedef struct {          /* thread identifier for Win32 */
    void *_Hnd; /* Win32 HANDLE */
    _Thrd_id_t _Id;
} _Thrd_imp_t;
typedef _Thrd_imp_t _Thrd_t;
```

2
```
// xthrcommon.h
class thread {
    // …
    _Thrd_t _Thr;
};
```

- A POSIX thread implementation

```
// POSIX thread
pthread_t
// …
```

Threads: under the Hood

```
class thread {
    public:
    // types
    class id; // the type of thread identifier
    // construct/copy/destroy
    thread() noexcept; // create a thread that does not (yet) have a task
    template<class F, class... Args> explicit thread(F&& f, Args&&... args);
    ~thread(); // if thread is joinable, then terminate
    // disable copy operations
    thread(const thread&) = delete;
    thread& operator=(const thread&) = delete;
    // threads are move-only
    thread(thread&&) noexcept;
    thread& operator=(thread&&) noexcept;
    // members
    void swap(thread&) noexcept; // exchange the values of threads
    bool joinable() const noexcept; // Is there a thread of execution associated with thread?
    void join(); // join it with the current thread
    void detach(); // Ensure that no system thread is associated with this thread object
    id get_id() const noexcept; // get the id of thread
    // static members
    static unsigned int hardware_concurrency() noexcept; // get the number of processing units
};
```

- The thread constructor is a variadic template

Threads: under the Hood

- Thread identity
- Hardware concurrency

ThreadManipTest

Prog.

The This_thread namespace

- Note the use of the this_thread namespace to disambiguate when you are requesting the id for the current thread, vs the id of a child thread. The get_id name for this action remains the same in the interest of reducing the conceptual footprint of the interface.

```
td::thread my_child_thread(f);
typedef std::thread::id ID;

ID my_id = std::this_thread::get_id();  // The current thread's id
ID your_id = my_child_thread.get_id();  // The child   thread's id
```

The Sleep_for function: an example

```cpp
#include <iostream>
#include <thread>
#include <chrono>

void foo() // simulate expensive operation
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

void bar() // simulate expensive operation
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

int main()
{
    std::cout << "starting first worker...\n";
    std::thread worker1(foo);

    std::cout << "starting second worker...\n";
    std::thread worker2(bar);

    std::cout << "waiting for workers to finish..." << std::endl;
    worker1.join(); worker2.join();

    std::cout << "done!\n";
}
```

C++ thread class: definitions

- The class thread provides a mechanism to create a new thread of execution, to join with a thread (i.e., wait for a thread to complete), and to perform other operations that manage and query the state of a thread.

C++ committee draft, section 33.3.2.p1 [thread.thread.class] *page 1304.*

- The class thread represents *a single thread of execution*. Threads allow multiple functions to execute concurrently.

https://en.cppreference.com/w/cpp/thread/thread

cppreference.com

Memory model

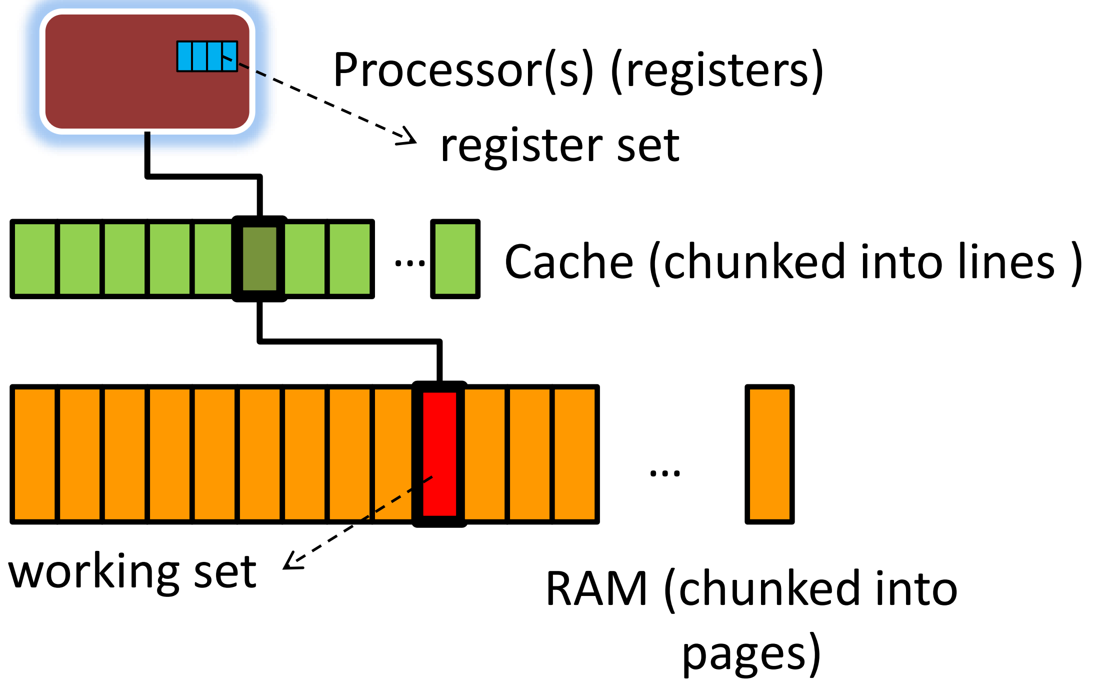
C++ concurrency mechanisms = Memory model (Core language) + Standard library components

- Memory model: set of language guarantees
- Memory model as a contract between *machine architect* and *compiler writers*: A memory model is an agreement between machine architects and compiler writers about how best to represent computer hardware.
- Memory model as a contract between *implementers* and *programmers*: A memory model is an agreement between the implementers and programmers to ensure that most programmers do not have to think about the details of computer hardware.

Memory model cont.

- Remember:

Operations on an object in memory are never directly performed on the object in memory. Instead, the object is loaded into a processor register, modified there, and then written back. Worse still, an object is typically first loaded from the main memory into a cache memory and from there to a register.

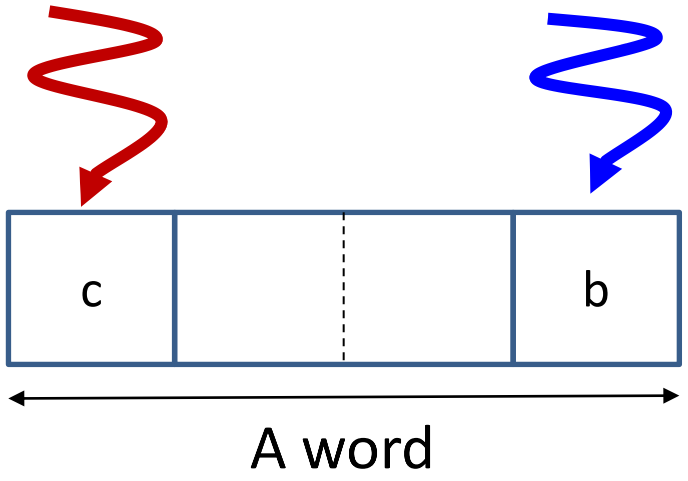


```
// add one to x (very simplified pseudo code)
load x into cache element Cx
    load Cx into register Rx
        Rx = Rx + 1
    store Rx back into Cx
store Cx back into x
```

Memory location

- An example

```
// thread 1
char c = 0;
void f()
{
      c = 1;
      int x = c;
      // x == 1 ??
}
```



```
// thread 2
char b = 0;
void f()
{
      b = 1;
      int y = b;
      // y == 1 ??
}
```

- The C++ memory model guarantees that two threads of execution can update and access <u>separate memory locations</u> without interfering with each other.

Bit-field

- Declares a class data member with explicit size, in bits. Adjacent bit field members may be packed to share and straddle the individual bytes.

1

```
struct S {
     // three-bit unsigned field,
     // allowed values are 0...7
     unsigned int b : 3;
};
void f()
{
     S s = {6};
    ++s.b; // store the value 7 in the bit field
     ++s.b; // the value 8 does not fit in this bit field,
     // s.b == 0
}
```

2

```
struct S {
    // will usually occupy 2 bytes:
    // 3 bits: value of b1
    // 2 bits: unused
    // 6 bits: value of b2
    // 2 bits: value of b3
    // 3 bits: unused
    unsigned char b1 : 3, : 2, b2 : 6, b3 : 2;
};
```

- Bit-fields give access to parts of the word.

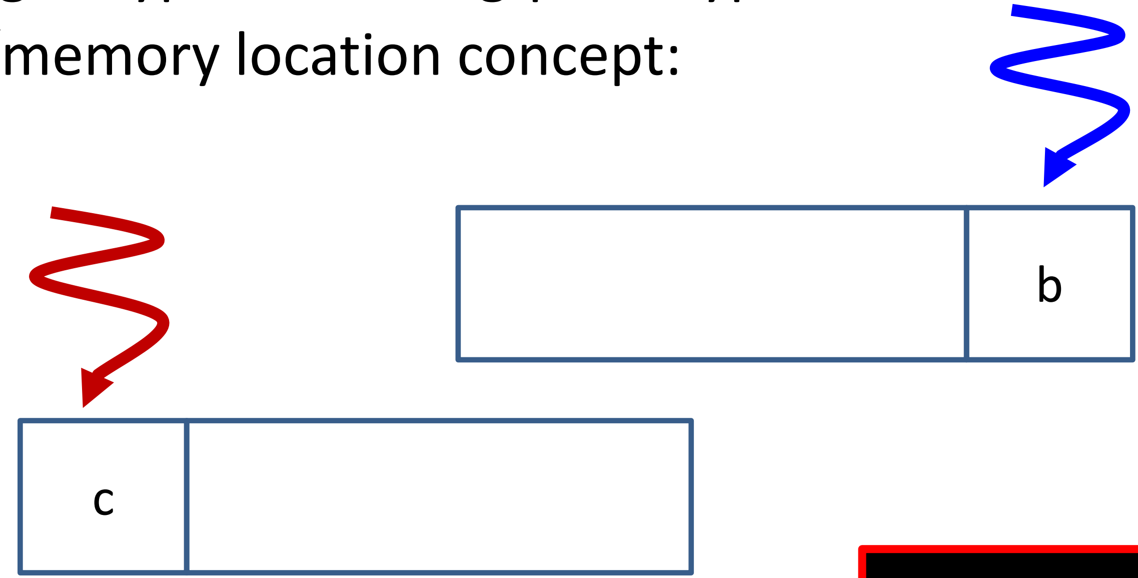
3

```
struct S {
    // will usually occupy 2 bytes:
    // 3 bits: value of b1
    // 5 bits: unused
    // 6 bits: value of b2
    // 2 bits: value of b3
    unsigned char b1 : 3;
    unsigned char :0; // start a new byte
    unsigned char b2 : 6;
    unsigned char b3 : 2;
};
```

Memory location cont.

- A *memory location* is either an object of arithmetic type, a pointer, or a maximal sequence of adjacent bit-field all having nonzero width.
- Arithmetic types: integral types + floating-point types
- With memory model/memory location concept:

```
// thread 1
char c = 0;
void f()
{
      c = 1;
      int x = c;
      // x == 1
}
```

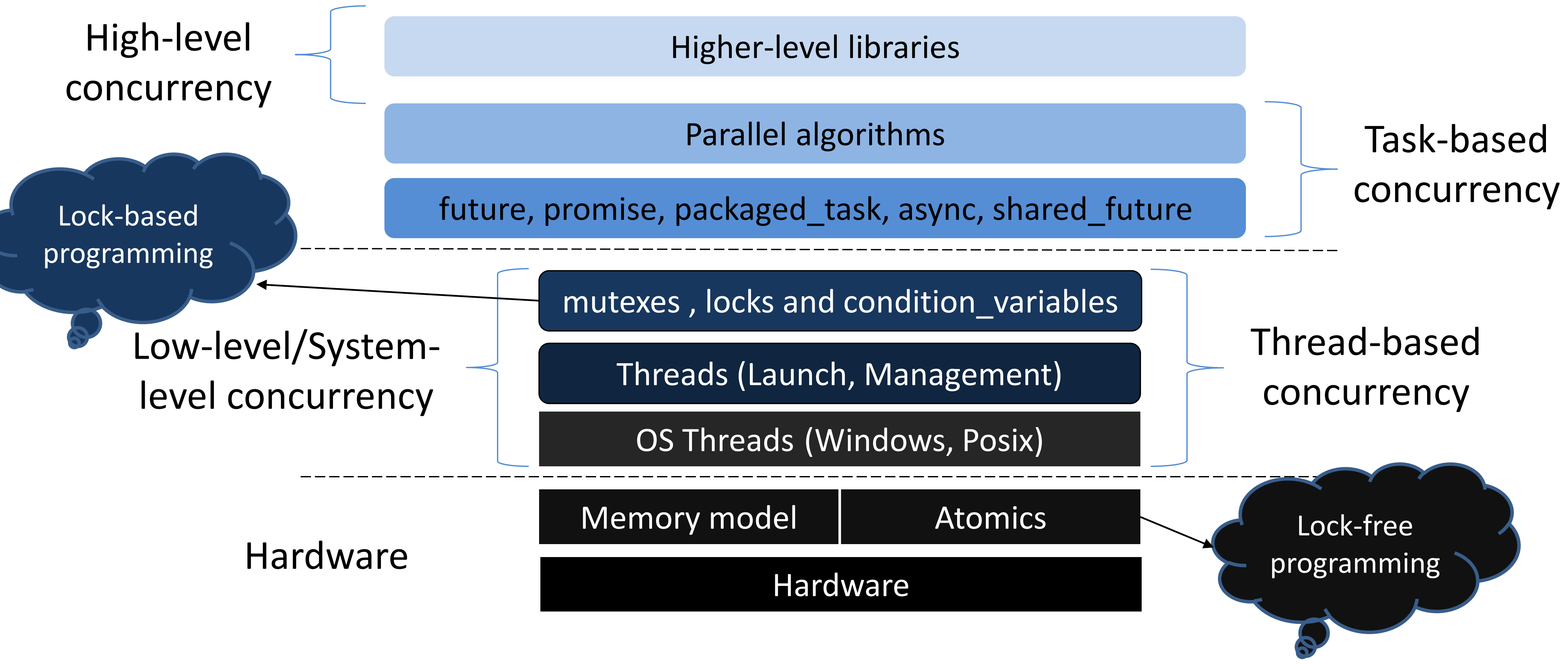


```
// thread 2
char b = 0;
void f()
{
      b = 1;
      int y = b;
      // y == 1
}
```

- Update bit-fields from separate threads with explicit synchronization.

```
struct S {
      char a; // location #1
      int b: 5; // location #2
      unsigned c: 11;
      unsigned :0; // 0 is "special"
      unsigned d:8; // location #3
      struct { int ee: 8; } e; // location #4
};
```

Modern C++ concurrent programming facilities



Instruction Re-ordering

- To gain performance, compilers, optimizers, and hardware reorder instructions.

Reordering
- Write reordering
- Read reordering

- Changing the meaning of your program

```
// thread 1
int x;
bool b;

void init()
{
     x = init(); // no use of b in initialize()
     b = true;
     // …
}
```

```
// thread 2
extern int x;
extern bool b;

void f()
{
     int y;
     while (!b) // if necessary,
                // wait for initialization to complete
          this_thread::sleep_for(milliseconds{10});
     y = x;
     // …
}
```

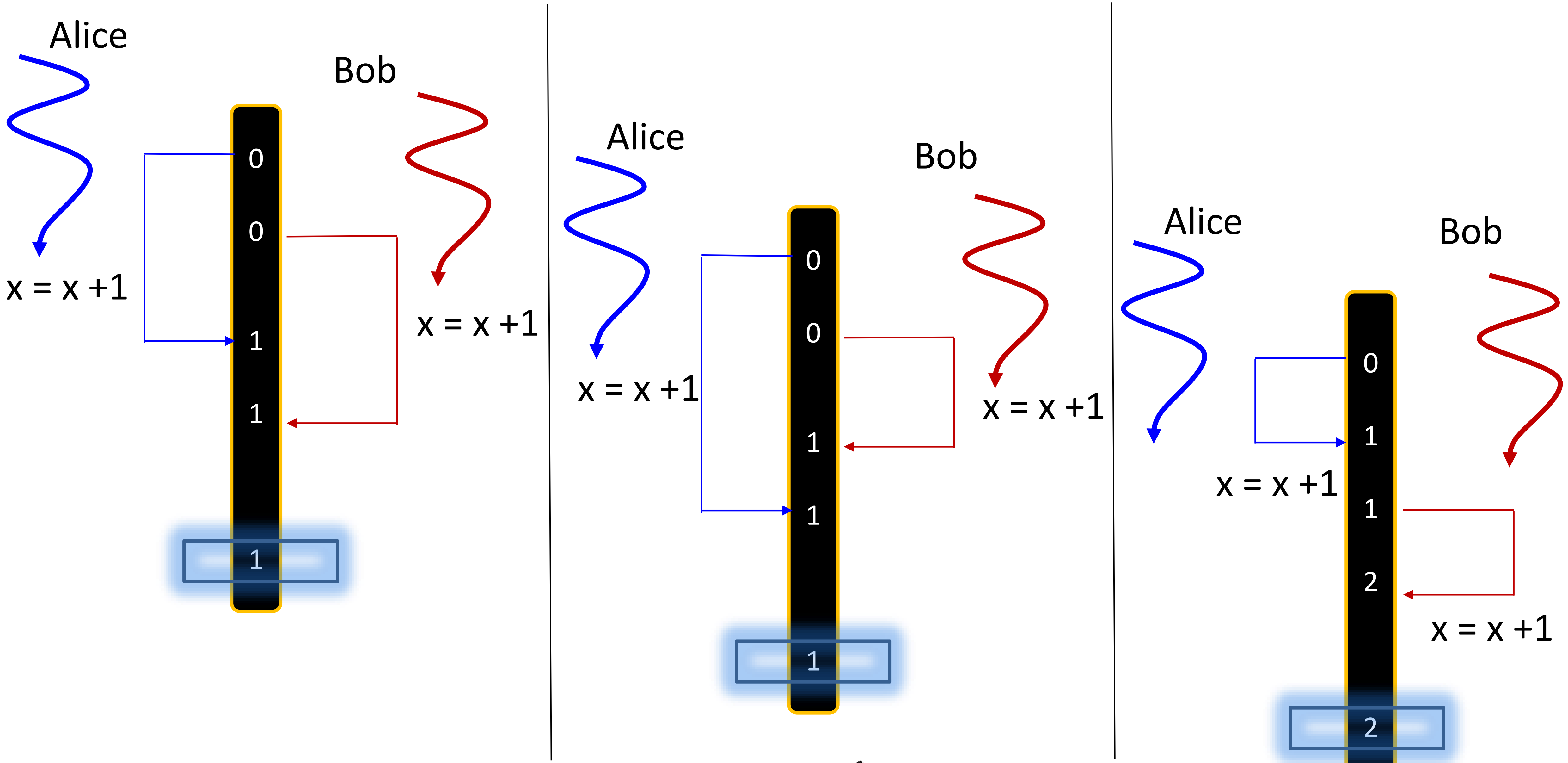
Memory order and sequential consistency

- The execution of a multithreaded program can be viewed as follows: Each step in the execution consists of choosing one of the threads to execute next, and then performing the next step in its execution. This process is repeated until the program as a whole terminates.
- Convoluted execution
- *Memory ordering* is the term used to describe what a programmer can assume about what a thread sees when it looks at a value from memory.
- L. Lamport definition:

A program is sequentially consistent if:

- the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and
- the operations of each individual processor were to appear in this sequence in the order specified by its program.

Memory order: an example



Data Race: A few examples

```
// thread t1 (or processor p1)
int data = 0;
flag = true;
                                // thread t2 (or processor p2)
                                while (!flag) {}
                                int copy_of_data = data;
```

from Mark D. Hill. Multiprocessors Should Support Simple Memory-Consistency Model. *Computer* 31(8), August 1998. pages 28-34.

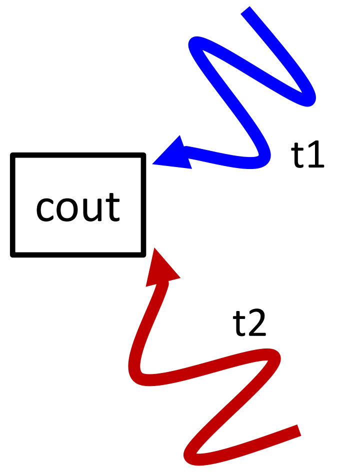
- flag is used for synchronization. Its purpose is to coordinate accesses to data.

```
bool flag{true};
void work() {
  while (flag) { std::this_thread::sleep(t); }
}
int main() {
  std::thread worker{work};
  std::cin.ignore();
  flag = false;
  worker.join();
}
```

Threads and Shared objects: an example

- Concurrent writing in output

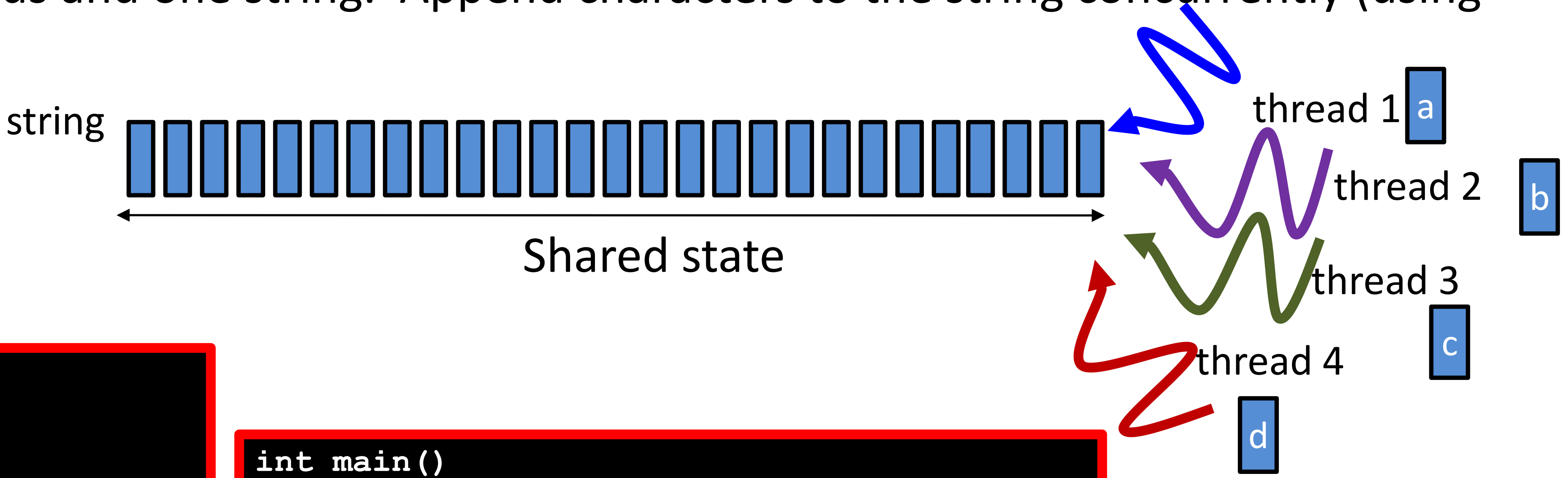
```
#include <thread>
#include <iostream>
void f1();
void f2();

int main()
{
  std::thread t1{f1};
  std::thread t2{f2};
  t1.join();
  t2.join();
  return 0;
}
void f1() { std::cout << "Hello "; }
void f2() { std::cout << "Parallel, world!\n"; }
```



- cout: shared object
- Data race
- Synchronization

Concurrent string appending

- We have several threads and one string. Append characters to the string concurrently (using multithreading)



- No synchronization

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

std::string s; // shared variable
void concatenate(char c, int n)
{
    for (int i = 0; i < n; ++i)
        s += c;
}
```

```
int main()
{
    std::thread cat1{concatenate, 'a', 1};
    std::thread cat2{concatenate, 'b', 2};
    std::thread cat3{concatenate, 'c', 3};
    std::thread cat4{concatenate, 'd', 4};

    cat1.join();  cat2.join();
    cat3.join();  cat4.join();

    std::cout << s << '\n';
    return 0;
}
```

Concurrent string appending with synchronization

```
#include <string>
#include <thread>
#include <mutex>
#include <iostream>

std::string s; // shared variable
std::mutex m;
void concatenate(char c, int n)
{
    for (int i = 0; i < n; ++i) {
        m.lock();
        s += c;
        m.unlock();
    }
}
```

```
int main()
{
  thread cat1{concatenate, 'a', 1};
  thread cat2{concatenate, 'b', 2};
  thread cat3{concatenate, 'c', 3};
  thread cat4{concatenate, 'd', 4};

  cat1.join();
  cat2.join();
  cat3.join();
  cat4.join();

  cout << s '\n';

  return 0;
}
```

What is a Mutex?

https://stackoverflow.com/questions/34524/what-is-a-mutex

When I am having a big heated discussion at work, I use a rubber chicken which I keep in my desk for just such occasions. The person holding the chicken is the only person who is allowed to talk. If you don't hold the chicken you cannot speak. You can only indicate that you want the chicken and wait until you get it before you speak. Once you have finished speaking, you can hand the chicken back to the moderator who will hand it to the next person to speak. This ensures that people do not speak over each other, and also have their own space to talk.

Replace Chicken with Mutex and person with thread and you basically have the concept of a mutex.

Threads Synchronization: Mutex

- Mutex: A Mutual Exclusive object.
- A mutex is an object used to represent *exclusive access* to some resource.

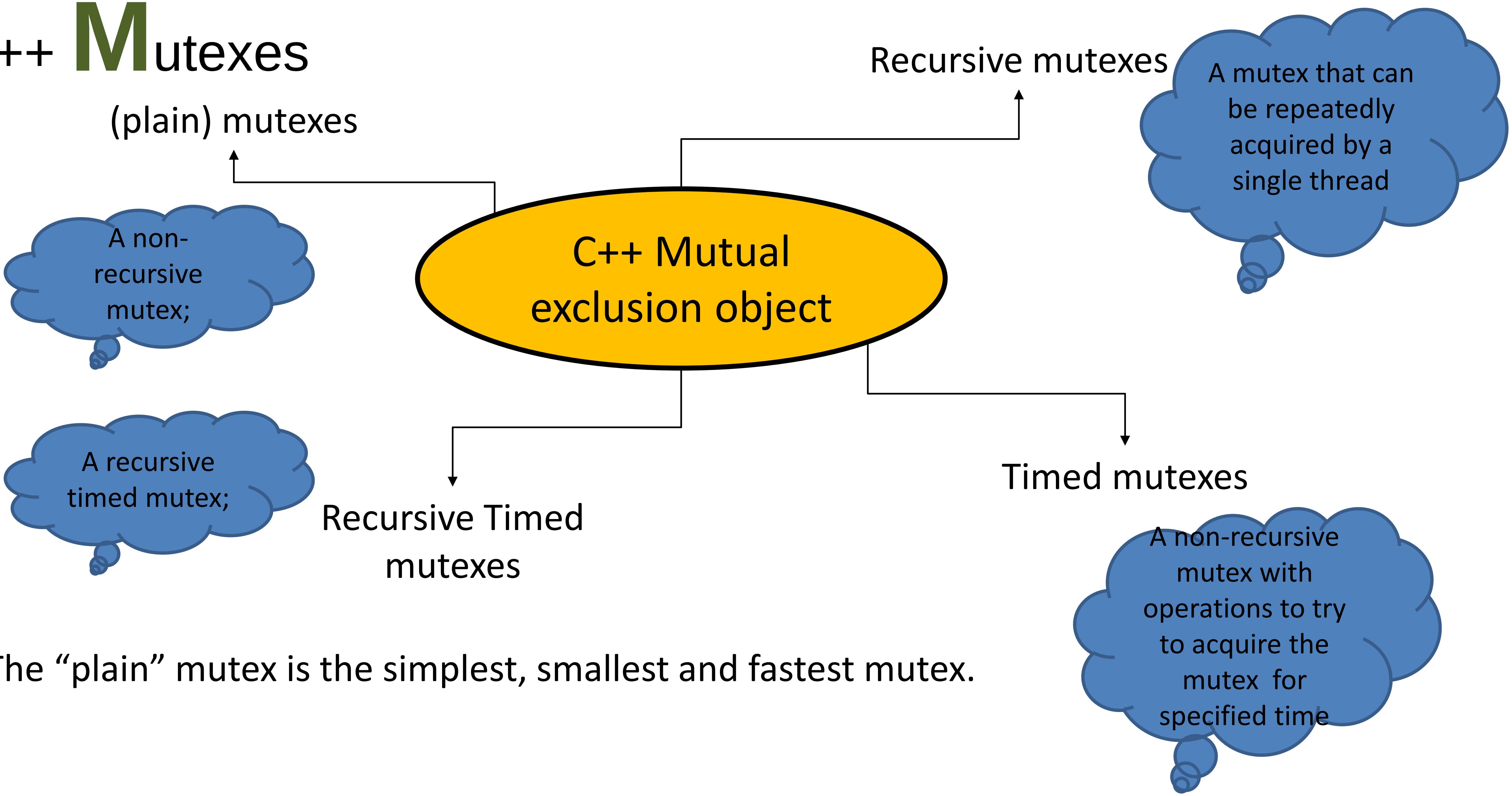
Mutex
- To protect against data races
- To synchronize access to data shared between multiple threads

- Only one thread can own a mutex at any one time.

Mutex operations
- To *acquire* a mutex: to *gain* exclusive ownership → To block an executing thread
- To *release* a mutex: to *relinquish* exclusive ownership → To unblock waiting threads

ALPHA آلفا

C++ Mutexes


C++ Mutual exclusion object
(plain) mutexes
A non-recursive mutex;
Recursive mutexes
A mutex that can be repeatedly acquired by a single thread
Timed mutexes
A non-recursive mutex with operations to try to acquire the mutex for specified time
Recursive Timed mutexes
A recursive timed mutex;

- The “plain” mutex is the simplest, smallest and fastest mutex.

class Mutex: details

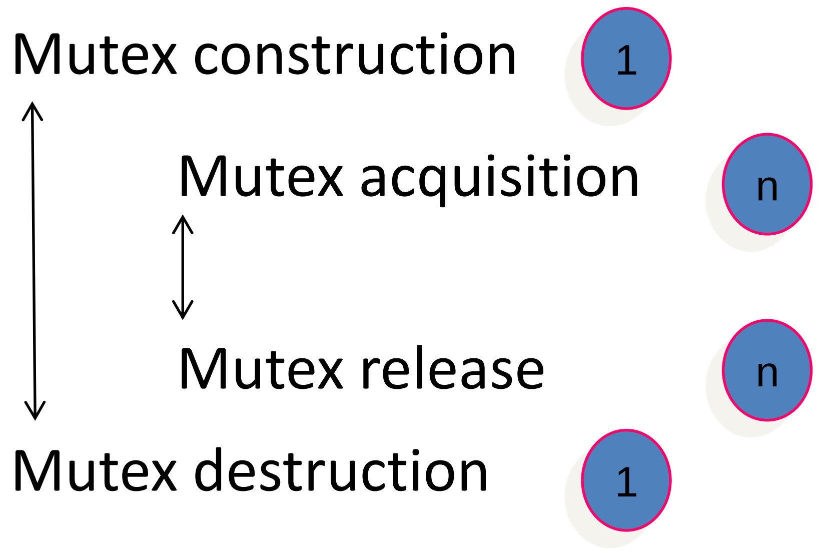
Exclusive ownership semantics: If one thread owns a mutex object, attempts by another thread to acquire ownership of that object will fail (for try_lock()) or block (for lock()) until the owning thread has released ownership with a call to unlock().

```
class mutex {
public:
    // constructors
    constexpr mutex() noexcept; // default ctor: mutex is not owned by any thread
    // destructor
    ~mutex();
    // disable copy operations
    mutex(const mutex&) = delete;
    mutex& operator=(const mutex&) = delete;
    // main operations
    void lock(); // acquire mutex, block
    bool try_lock(); // try to acquire mutex, did acquisition proceed?
    void unlock(); // release the mutex
};
```

- A mutex cannot be copied or moved.
- Think of a mutex as a resource, rather than a handle of a resource.

Mutexes: problems

- Mutex is a lock.
- A lock is a resource and it should be released.



```
void use(mutex& mtx, vector<string>& vs, int i)
{
    mtx.lock();
    if ( i < 0) return; // forget to release mutex
    string s = v[i]; // exception throw
    // …
    mtx.unlock();
}
```

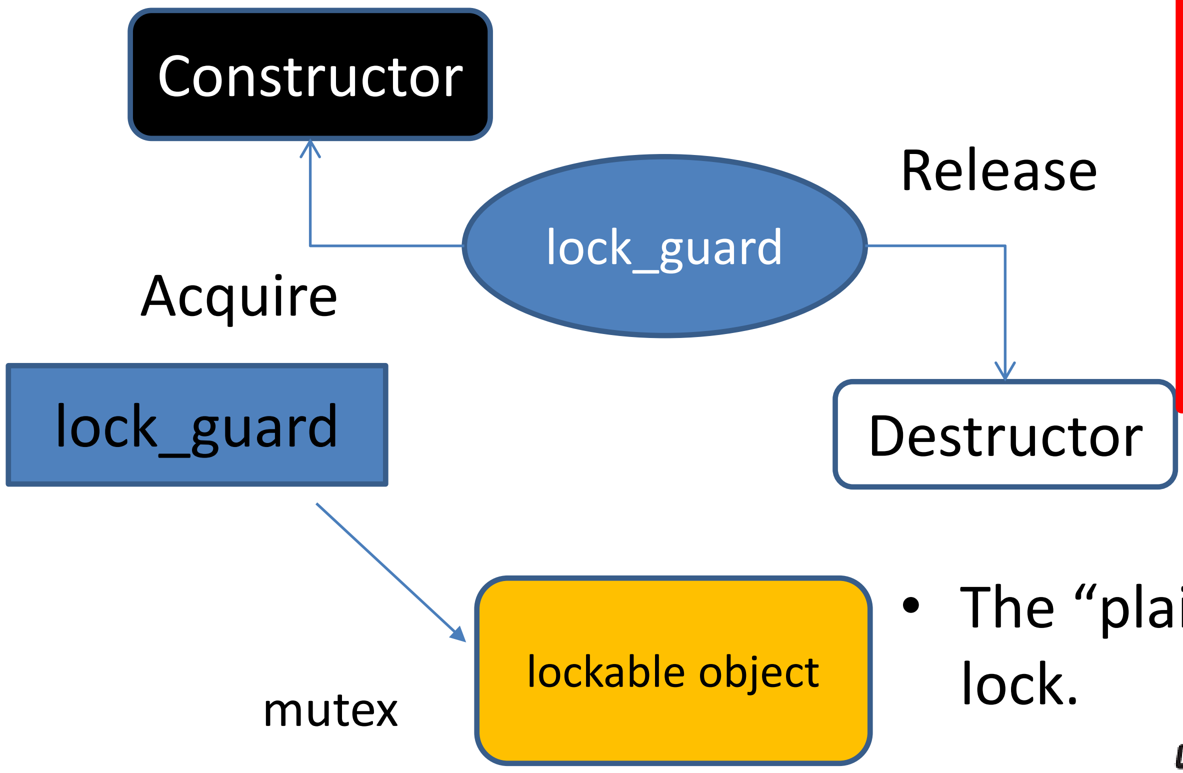
- Each lock operation should be matched by an unlock operation.

- The standard library provides two RAII classes, lock_guard and unique_lock to handle such problems.

Lock_guard

A *lock is an object that holds a reference to a lockable object and may unlock the lockable object during the* lock's destruction (such as when leaving block scope).

- Locks: lock_guard and unique_lock
- RAII: Resource Acquisition Is Initialization



```
// <mutex> mutex header
template<class Mutex>
class lock_guard {
      using mutex_type = Mutex;
      Mutex& m_;
public:
      explicit lock_guard(Mutex&);
      lock_guard(mutex_type& m, adopt_lock_t);
      ~lock_guard();
      lock_guard(const lock_guard&) = delete;
      lock_guard& operator=(const lock_guard&) = delete;
      // …
};
```

- The "plain" lock_guard is the simplest, smallest and fastest lock.

Lock_guard: an example

- Good

```
void use(mutex& mtx, vector<string>& vs, int i)
{
    lock_guard<mutex> g{mtx};
    if ( i < 0) return; // forget to release mutex
    string s = v[i]; // exception throw
} // calling g's destructor: unlock the mutex
```

- Better: Minimize the size of critical section

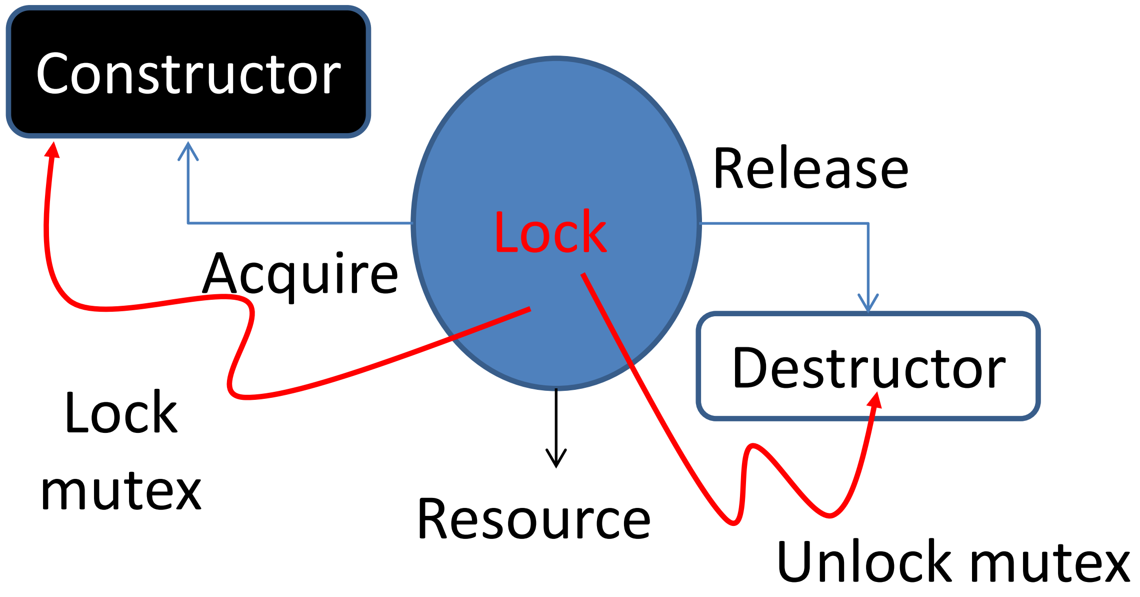
```
void use(mutex& mtx, vector<string>& vs, int i)
{

    if (i < 0) return; // forget to release mutex
     lock_guard<mutex> g{mtx};
    string s = v[i]; // exception throw
} // calling g's destructor: unlock the mutex
```

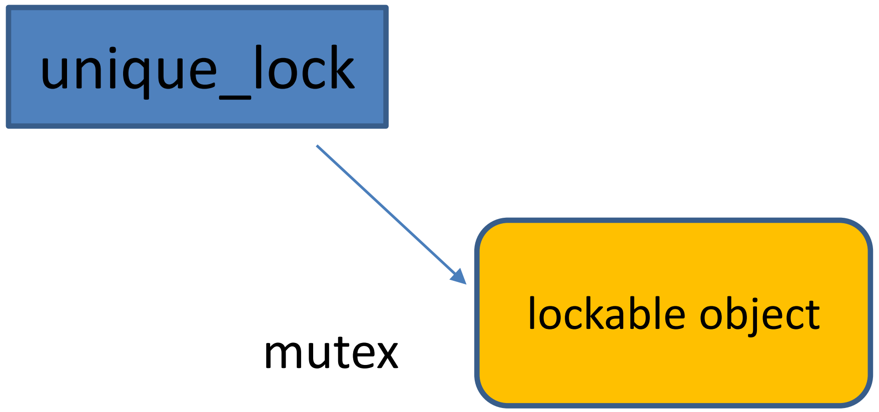
Code review

MutexWithWorkerThreadsTest Prog.

LockGuardWithWorkingThreadsTest Prog.

Unique_lock


Constructor
Lock
Release
Acquire
Destructor
Lock
mutex
Resource
Unlock mutex


unique_lock
mutex
lockable object

Unique_lock: details

```
template <class Mutex>
class unique_lock {
public:
    using mutex_type = Mutex;
    unique_lock() noexcept; // default ctor: lock does not hold a mutex
    explicit unique_lock(Mutex&); // acquire mutex
    template<class Clock, class Duration>
    unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time); // lock until
    template<class Rep, class Period>
    unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);  // lock for
    // disable copy operations
    unique_lock(unique_lock const&) = delete;
    unique_lock& operator=(unique_lock const&) = delete;
    unique_lock(unique_lock&& u) noexcept;
    unique_lock& operator=(unique_lock&&) noexcept;
    ~unique_lock(); // release mutex
    void lock(); // lock the mutex
    bool try_lock(); // call mutex try_lock: did acquisition succeed?
    void unlock(); // unlock the mutex
    mutex_type* release() noexcept;
    bool owns_lock() const noexcept;
    // observers
    explicit operator bool () const noexcept;
    mutex_type* mutex() const noexcept;
private:
    Mutex *pm;  bool owns;
};
```

unique_lock vs. lock_guard cont.

- The “plain” lock_guard is the simplest, smallest and fastest guard.
- Use lock_guard, if you need to just simple RAII locking facility.
- Use unique_lock, if you need to RAII locking facility and operations on a contained mutex.

Condition variables

- Condition variables are used to manage communication among threads.
- A thread can wait (block) on a condition variable until some event, such as reaching specific time, or another thread completing occurs.
- Blocking means *Waiting*.

Condition variables provide synchronization primitives used to block a thread until notified by some other thread that some condition is met or until a system time is reached.

- Condition variables: condition_variable and condition_variable_any
- Class condition_variable provides a condition variable that can only wait on an object of type unique_lock, allowing maximum efficiency on some platforms.

class Condition_variable

- The condition_variable class is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until another thread both modifies a shared variable (the condition), and notifies the condition_variable.
- notify_one vs. notify_all:

  notify_one: If any threads are blocked waiting for this condition variable, unblocks one of those threads → wakes up only one blocking thread

  notify_all: Unblocks all threads that are blocked waiting for the this condition variable → wakes up all blocking threads

- notify_one is more efficient

```
class condition_variable {
public:
  void notify_one() noexcept;
  void notify_all() noexcept;
  void wait(unique_lock<mutex>& lock);
  // …
};
```

- wait: Atomically calls lock.unlock() and blocks the thread on the condition variable.

How Condition variables work?

- The thread that intends to modify the variable has to
  1- acquire a std::mutex (typically via std::unique_lock)
  2- perform the modification while the lock is held
  3- execute notify_one or notify_all on the std::condition_variable (the lock does not need to be

- Any thread that intends to wait on std::condition_variable has to
  1- acquire a std::unique_lock<std::mutex>, on the same mutex as used to protect the shared variable
  2- execute wait, wait_for, or wait_until. The wait operations atomically release the mutex and suspend the execution of the thread.
  3- When the condition variable is notified, a timeout expires, or a spurious wakeup occurs, the thread is awakened, and the mutex is atomically reacquired. The thread should then check the condition and resume waiting if the wake up was spurious.

Condition variable- an example

Condition VariableTest Prog.

Condition_variable: details

```
class condition_variable {
public:
    condition_variable();
    ~condition_variable(); // destructor: no thread may be waiting and not notified
    // prohibit copy operations
    condition_variable(const condition_variable&) = delete;
    condition_variable& operator=(const condition_variable&) = delete;
    void notify_one() noexcept; // unblock one waiting thread
    void notify_all() noexcept; // unblock all waiting threads
    void wait(unique_lock<mutex>& lock); // atomically call lock.unlock(), and blocks
    template<class Predicate>
    void wait(unique_lock<mutex>& lock, Predicate pred); // while (!pred()) wait(lock)
    template<class Clock, class Duration>
    cv_status wait_until(unique_lock<mutex>& lock, const chrono::time_point<Clock, Duration>& abs_time);
    template<class Clock, class Duration, class Predicate>
    bool wait_until(unique_lock<mutex>& lock,
    const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);
    template<class Rep, class Period>
    cv_status wait_for(unique_lock<mutex>& lock,
    const chrono::duration<Rep, Period>& rel_time);
    template<class Rep, class Period, class Predicate>
    bool wait_for(unique_lock<mutex>& lock,
    const chrono::duration<Rep, Period>& rel_time,
    Predicate pred);
```

Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant

Q & A