Use a more efficient algorithm than the copy-and-sort version in the slides.

```
#include <type_traits>
#include <iostream>
#include <utility>

template <typename T>
T max2(T && t)
{
    return t;
}

template <typename T, typename... Args>
typename std::common_type<T, Args...>::type
max2(T && t, Args&& ... args)
{
    typedef typename std::common_type<T, Args...>::type C;
    return std::max(t, max2(std::forward<C>(args)...));
}

int main10()
{
    std::cout << max2(5, 14u, 22.1, 9.3, 13) << std::endl;
    return 0;
}
```

Basic concurrency

1) Using std::async, open multiple files in parallel and count the total number of characters across all the files

```
#include <iostream>
#include <fstream>
#include <thread>
#include <future>
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>

std::size_t countCharsInFile(const std::string & fileName)
{
    std::ifstream input(fileName);
    if (! input)
        std::cout << "Can't open " << fileName << std::endl;
    return std::distance(std::istreambuf_iterator<char>(input),
                         std::istreambuf_iterator<char>());
}

int main(int argc, char * argv[])
{
    std::vector<std::future<std::size_t>> futures;
    for (auto i = 1; i != argc; ++i) {
        auto f = std::async(std::launch::async, countCharsInFile, argv[i]);
        futures.push_back(std::move(f));
    }
```

```
    std::size_t total = 0;
    for (auto & f : futures)
        total += f.get();

    std::cout << "Total characters = " << total << std::endl;
    return 0;
}
```

2) Fill a vector with random integers in the range 1 to 1000. Count the number of odd integers using the optimum number of threads

```
#include <random>
#include <vector>
#include <thread>
#include <algorithm>
#include <iostream>
#include <numeric>
#include <iterator>
#include <future>

int sum(std::vector<int>::const_iterator first,
std::vector<int>::const_iterator last)
{
    return std::accumulate(first, last, 0);
}

int main2()
{
    using namespace std;
    default_random_engine engine;
    uniform_int_distribution<> ints(1, 1000);

    std::vector<int> v;
    const auto numElems = 5000u;
    const auto minGrain = 1000u;
    const auto numThreads = min(numElems / minGrain,
                                thread::hardware_concurrency());
    generate_n(back_inserter(v), numElems, [&]{ return ints(engine); });
    const auto chunk = numElems / numThreads;
    std::vector<future<int>> futures;
    for (auto i = 0; i != numThreads; ++i) {
        const auto begin = v.cbegin()+i*chunk;
        const auto end = v.cbegin()+min((i+1)*chunk, v.size());
        auto f = async(launch::async, sum, begin, end);
        futures.push_back(std::move(f));
    }
    int total = 0;
    for (auto & f : futures)
        total += f.get();
    cout << "Total = " << total << endl;
    return 0;
}
```

3) Implement a simple spin lock using an atomic

```
#include <atomic>
#include <future>
#include <vector>
#include <iostream>

std::atomic<bool> spinlock(false);

void lock_unlock()
{
    for (auto i = 0; i != 10'000; ++i) {
        bool expect = false;
        while (! spinlock.compare_exchange_weak(expect, true)) {
            expect = false;
        }
        std::cout << "got ";
        std::cout << "the ";
        std::cout << "lock\n";
        spinlock = false;
    }
}

int main3()
{
    const int numThreads = 10;

    std::vector<std::future<void>> futures;
    for (auto i = 0; i != numThreads; ++i)
        futures.emplace_back(std::async(std::launch::async, lock_unlock));
}
```

4) Initialise a global shared object pointed to by an atomic pointer and make sure the object is correctly deleted when the program exits. Do this without using any locks

```
#include <atomic>
#include <iostream>
#include <future>
#include <vector>

struct X { int stuff[100]; };

std::atomic<X*> xp{nullptr};

void initxp()
{
    if (xp == nullptr) {
        X * newX = new X;
        X * expected = nullptr;
        auto updated = xp.compare_exchange_strong(expected, newX);
        if (! updated) {
            std::cout << "didn't get it\n";
            delete newX;
        } else {
            std::cout << "got it\n";
        }
    } else {
        std::cout << "already assigned\n";
```

```
    }
}

int main4()
{
    {
        const int numThreads = 10;
        std::vector<std::future<void>> futures;
        futures.reserve(numThreads);
        for (auto i = 0; i != numThreads; ++i)
            futures.emplace_back(std::async(std::launch::async, initxp));
    }
    delete xp;
}
```

5) Insert items at the head of a singly-linked list from multiple threads using a non-blocking algorithm using compare and exchange (CAS). Check that the correct number of items have been inserted into the list and that the list structure is correct

```
#include <iostream>
#include <atomic>
#include <thread>
#include <vector>

const auto numLoopsPerThread = 10 * 1000;
const auto numThreads = 4;

struct Node {
    Node * next;
    std::thread::id who;
};

std::atomic<Node *> head(nullptr);

void addToList()
{
    for (auto i = 0; i != numLoopsPerThread; ++i) {
        Node * newNode = new Node;
        newNode->next = head;
        newNode->who = std::this_thread::get_id();
#if 0
        head = newNode; // not thread safe!
#else
        while (! head.compare_exchange_weak(newNode->next, newNode))
            std::cout << "looping on head\n";
#endif
    }
}

int main5()
{
    std::vector<std::thread> threads;
    for (auto i = 0; i != numThreads; ++i)
        threads.emplace_back(addToList);
    for (auto & t : threads)
        t.join();

    int nodeCount = 0;
    while (head.load()) {
```

```
        nodeCount++;
        Node * p = head;
        head = head.load()->next;
        delete p;
    }

    std::cout << "Node count = " << nodeCount << std::endl;

    return 0;
}
```

6) Create a single-producer multi-consumer queue based on a circular buffer. Use atomics to hold the positions for the writer and the readers. Use non-blocking queue operations throughout (such as spin locks or polling loops)

```
#include <iostream>
#include <atomic>
#include <vector>
#include <thread>
#include <typeinfo>
#include <limits>
#include <utility>

//const auto numItemsToWrite = 30 * 1000;
const auto numItemsToWrite = 20;
//const auto bufSize = 10 * 1000;
const auto bufSize = 10;
const auto numReaders = 4;

struct Item { int value; };

Item buf[bufSize];

std::atomic<int> readerPosns[numReaders] = { 0 };
std::atomic<int> writerPosn = 0;

int minReaderPosn()
{
    int minPosn = std::numeric_limits<int>::max();
    for (auto i = 0; i != numReaders; ++i)
        minPosn = std::min(minPosn, readerPosns[i].load());
    return minPosn;
}

class Writer {
public:
    Writer() : posn(0) {}
    void fillBuffer() {
        for (auto i = 0; i != numItemsToWrite; ++i) {
            while (posn >= minReaderPosn() + bufSize) {
                std::cout << "Waiting to write" << std::endl;
                std::this_thread::sleep_for(std::chrono::milliseconds(1));
            }
            buf[posn % bufSize].value = posn;
            std::cout << "Wrote item " << posn << std::endl;
            posn++;
            writerPosn++;
        }
```

```
    }

private:
    int posn;
};

class Reader {
public:
    Reader(int id) : id(id), posn(0) {}
    void readBuffer() {
        for (auto i = 0; i != numItemsToWrite; ++i) {
            while (posn >= writerPosn) {
                std::cout << "Reader " << id << " waiting to read"
                          << std::endl;
                std::this_thread::sleep_for(std::chrono::milliseconds(1));
            }
            std::cout << "Reader " << id << " read item " << posn
                          << std::endl;
            readerPosns[id]++;
            posn++;
        }
    }

private:
    int id, posn;
};

int main6()
{
    std::vector<std::thread> readers;
    std::thread writer(&Writer::fillBuffer, Writer());
    for (auto i = 0; i != numReaders; ++i)
        readers.emplace_back(&Reader::readBuffer, Reader(i));
    for (auto & t : readers)
        t.join();
    writer.join();
    return 0;
}
```

More concurrency

1) Update the RUList to be thread-safe by locking the appropriate methods. Is this a sensible design approach?

```
#include <iostream>
#include <mutex>
#include <vector>
#include <string>
#include <algorithm>

template <typename T>
class RUList
{
public:
    RUList() = default;
    RUList(std::initializer_list<T> lst) : v(lst) {
        if (v.size() > maxSize)
```

```
            v.resize(maxSize);
    }

    RUList(const RUList & other) {
        std::lock_guard<std::mutex> other_guard(other.m);
        v = other.v;
    }

    RUList(RUList && other) {
        std::lock_guard<std::mutex> other_guard(other.m);
        v = std::move(other.v);
    }

    RUList & operator=(const RUList & rhs) {
        RUList temp(rhs);
        std::lock(m, temp.m);
        std::lock_guard<std::mutex> guard_m(m, std::adopt_lock);
        std::lock_guard<std::mutex> guard_t(temp.m, std::adopt_lock);
        std::swap(v, temp.v);
        return *this;
    }

    RUList & operator=(RUList && rhs) {
        std::lock(m, rhs.m);
        std::lock_guard<std::mutex> guard_m(m, std::adopt_lock);
        std::lock_guard<std::mutex> guard_r(rhs.m, std::adopt_lock);
        v = std::move(rhs.v);
        return *this;
    }

    template <typename T2>
    void add(T2 && name) {
        std::lock_guard<std::mutex> guard(m);
        auto i = std::find(begin(v), end(v), std::forward<T2>(name));
        if (i != end(v))
            v.erase(i);
        v.insert(v.begin(), std::forward<T2>(name));
        if (v.size() > maxSize)
            v.pop_back();
        print();
    }

    void print() const {
        std::lock_guard<std::mutex> guard(m);
        std::cout << "---------\n";
        for (const auto & i : v)
            std::cout << i << "\n";
    }

    const T & operator[](unsigned int i) const noexcept {
        std::lock_guard<std::mutex> guard(m);
        return v[i];
    }
private:
    std::vector<T> v;
    mutable std::mutex m;
    static constexpr unsigned int maxSize = 3;
};

RUList<std::string> f() { return RUList<std::string>{"one elem"}; }
```

```
int main()
{
    RUList<std::string> l{"a", "b", "c", "d"};
    l.add("foo");
    l.add("foo");
    l.add("bar");
    l.add("bar");
    l.add("wibble");
    l.add("xyzzy");
    l.add("wibble");

    RUList<std::string> l2(std::move(l));
    l.print();
    l2.add("123");

    RUList<std::string> x;
    x = f();
    std::cout << x[0] << std::endl;

    RUList<std::string> l3(l2);
    l2 = l3;
}
```

2) Create two bank accounts, each with its own associated lock. Transfer random amounts based on a normal distribution between the two accounts in both directions using several parallel threads. Ensure that the accounts are locked and unlocked correctly to avoid deadlock

3) Initialise the random number engine in the previous exercise using a hardware random number source from only one of the threads

```
#include <iostream>
#include <mutex>
#include <thread>
#include <random>
#include <vector>
#include <algorithm>

class Account {
public:
    Account() : balance(0.0) {}
    void deposit(double amount) { balance += amount; }
    void withdraw(double amount) { balance -= amount; }
    double getBalance() const { return balance; }
    std::mutex & getLock() { return m; }
    void lock() { m.lock(); }
    void unlock() { m.unlock(); }
private:
    double balance /* = 0.0 */;
    std::mutex m;
};

std::default_random_engine engine;
std::normal_distribution<> amounts(0.0, 100.0);
std::mutex dreMutex;
std::once_flag callOnceFlag;
```

```
void setup()
{
    std::random_device rd;
    engine.seed(rd());
}

void transfer(Account & a1, Account & a2)
{
    std::call_once(callOnceFlag, setup);

    const auto numTransfers = 1000;
    for (auto i = 0; i != numTransfers; ++i) {
        dreMutex.lock();
        auto amount = amounts(engine);
        dreMutex.unlock();
        std::lock(a1.getLock(), a2.getLock());
        std::lock_guard<std::mutex> guarda1(a1.getLock(), std::adopt_lock);
        std::lock_guard<std::mutex> guarda2(a2.getLock(), std::adopt_lock);
        if (amount > 0.0) {
            a1.withdraw(amount);
            a2.deposit(amount);
        } else if (amount < 0.0) {
            a2.withdraw(-amount);
            a1.deposit(-amount);
        }
    }
}

int main4()
{
    Account a1, a2;

    std::vector<std::thread> tthrds;
    const auto numThreads = 10;
    for (auto i = 0; i != numThreads; ++i)
        tthrds.emplace_back(transfer, std::ref(a1), std::ref(a2));
    std::for_each(tthrds.begin(), tthrds.end(),
                  std::mem_fn(&std::thread::join));
    std::cout << a1.getBalance() << " " << a2.getBalance() << std::endl;
    return 0;
}
```

4) Implement a thread-safe fixed-length queue using blocking operations (instead of spin locks or polling)

```
#include <iostream>
#include <deque>
#include <thread>
#include <mutex>
#include <condition_variable>

template <typename T>
class Queue {
public:
    void push(const T & t) {
        std::unique_lock<std::mutex> fullLock(queueMutex);
        full.wait(fullLock, [this]{ return queue.size() < maxQueueLen; });
        queue.push_back(t);
        std::cout << "Pushed item " << t << std::endl;
```

```
        empty.notify_one();
    }

    void pop(T & t) {
        std::unique_lock<std::mutex> emptyLock(queueMutex);
        empty.wait(emptyLock, [this]{ return ! queue.empty(); });
        t = std::move(queue.front());
        queue.pop_front();
        std::cout << " Popped item " << t << std::endl;
        full.notify_one();
    }

private:
    std::deque<T> queue;
    std::mutex queueMutex;
    std::condition_variable empty, full;
    static const int maxQueueLen = 4;
};

Queue<int> queue;

const auto numItemsToInsert = 30;

void writer()
{
    for (auto i = 0; i != numItemsToInsert; ++i)
        queue.push(i+1);
}

void reader()
{
    int item;
    for (auto i = 0; i != numItemsToInsert; ++i)
        queue.pop(item);
}

int main5()
{
    std::thread writeThread(writer);
    std::thread readThread(reader);
    writeThread.join();
    readThread.join();
    return 0;
}
```

5) Add push and pop methods that will time out if they block for longer than a given time. Refactor out any duplication

```
#include <iostream>
#include <deque>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <chrono>
```

```
template <typename T>
class Queue {
public:
    Queue() :
        emptyPred([this]{ return ! queue.empty(); }),
        fullPred([this]{ return queue.size() < maxQueueLen; }) {}

    void push(const T & t) {
        std::unique_lock<std::mutex> fullLock(queueMutex);
        full.wait(fullLock, fullPred);
        queue.push_back(t);
        std::cout << "Pushed item " << t << std::endl;
        empty.notify_one();
    }

    void push_wait_for(const T & t, const std::chrono::milliseconds & dur) {
        std::unique_lock<std::mutex> fullLock(queueMutex);
        auto ok = full.wait_for(fullLock, dur, fullPred);
        if (ok) {
            queue.push_back(t);
            std::cout << "Pushed item " << t << std::endl;
            empty.notify_one();
        } else {
            std::cout << "*** Waited too long\n";
        }
    }

    void pop(T & t) {
        std::unique_lock<std::mutex> emptyLock(queueMutex);
        empty.wait(emptyLock, emptyPred);
        t = std::move(queue.front());
        queue.pop_front();
        std::cout << " Popped item " << t << std::endl;
        full.notify_one();
    }

private:
    std::deque<T> queue;
    std::mutex queueMutex;
    std::condition_variable empty, full;
    std::function<bool()> emptyPred, fullPred;
    static const int maxQueueLen = 4;
};

Queue<int> queue;

const auto numItemsToInsert = 30;

void writer()
{
    for (auto i = 0; i != numItemsToInsert; ++i)
        queue.push_wait_for(i+1, std::chrono::milliseconds(1));
}

void reader()
{
    int item;
    for (auto i = 0; i != numItemsToInsert; ++i)
        queue.pop(item);
}
```

```
int main6()
{
    std::thread writeThread(writer);
    writeThread.join();
    return 0;
}
```

6) Add appropriate memory ordering flags to the circular buffer queue from the previous section

```
#include <iostream>
#include <atomic>
#include <vector>
#include <thread>
#include <typeinfo>
#include <limits>
#include <utility>

const auto numItemsToWrite = 20;
const auto bufSize = 10;
const auto numReaders = 4;

struct Item { int value; };

Item buf[bufSize];

std::atomic<int> readerPosns[numReaders];
std::atomic<int> writerPosn;

int minReaderPosn()
{
    int minPosn = std::numeric_limits<int>::max();
    for (auto i = 0; i != numReaders; ++i)
        minPosn = std::min(minPosn,
                           readerPosns[i].load(std::memory_order_relaxed));
    return minPosn;
}

class Writer {
public:
    Writer() : posn(0) {}
    void fillBuffer() {
        for (auto i = 0; i != numItemsToWrite; ++i) {
            while (posn >= minReaderPosn() + bufSize) {
                std::cout << "Waiting to write" << std::endl;
                std::this_thread::sleep_for(std::chrono::milliseconds(1));
            }
            buf[posn % bufSize].value = posn;
            std::cout << "Wrote item " << posn << std::endl;
            posn++;
            writerPosn.fetch_add(1, std::memory_order_release);
        }
    }

private:
    int posn;
};
```

```
class Reader {
public:
    Reader(int id) : id(id), posn(0) {}
    void readBuffer() {
        for (auto i = 0; i != numItemsToWrite; ++i) {
            while (posn >= writerPosn.load(std::memory_order_acquire)) {
                std::cout << "Reader " << id << " waiting to read"
                        << std::endl;
                std::this_thread::sleep_for(std::chrono::milliseconds(1));
            }
            std::cout << "Reader " << id << " read item " << posn
                        << std::endl;
            posn++;
            readerPosns[id].fetch_add(1, std::memory_order_release);
        }
    }

private:
    int id, posn;
};

int main7()
{
    std::fill(std::begin(readerPosns), std::end(readerPosns), 0);
    writerPosn = 0;
    std::vector<std::thread> readers;
    std::thread writer(&Writer::fillBuffer, Writer());
    for (auto i = 0; i != numReaders; ++i)
        readers.emplace_back(&Reader::readBuffer, Reader(i));
    for (auto & t : readers)
        t.join();
    writer.join();
    return 0;
}
```