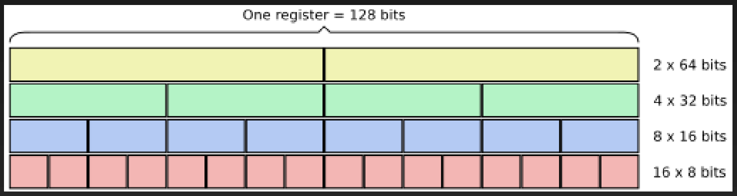
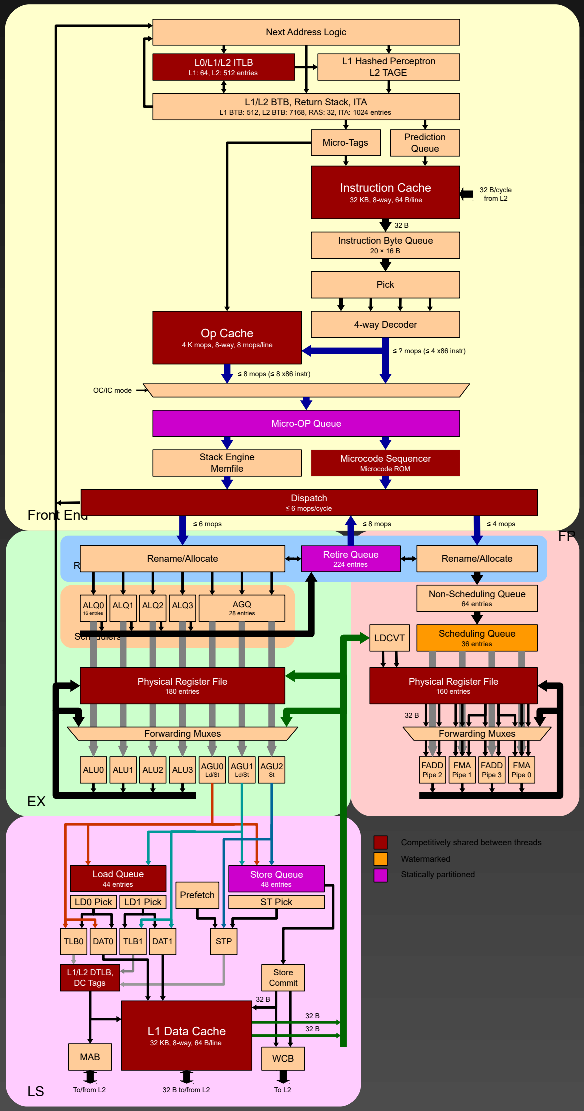
Concurrent vs. Parallel vs. Distributed

- Concurrent
  - different parts of the program running in single system at the same time, can communicate
- Parallel
  - single computation divided to smaller and same parts, running concurrently
- Distributed
  - concurrent execution on different computers

HW Parallelism

- Registers
  - SSE = 128b, AVX(2) = 256b, AVX512, …
  - parts of 2^n bytes
- Instructions
  - more ALUs
  - Out Of Order (OOO) execution
- SMT = Simultaneous Multi-Threading
  - more threads (2-16) in single core
  - better ALU utilization
- SMP – Symmetric Multi-Processing
- Cluster – computers + fabric (FAST interconnect)
- Grid, Cloud – servers + common network





Shared Data

- Possible Race Condition
  - Conflict over a resource without coordination
    - Bad things happen as a result – Undefined Behaviour
- Data Race Definition
  1) Two or more threads access the same memory
  2) At least one access is a write
  3) The threads do not synchronize with each other

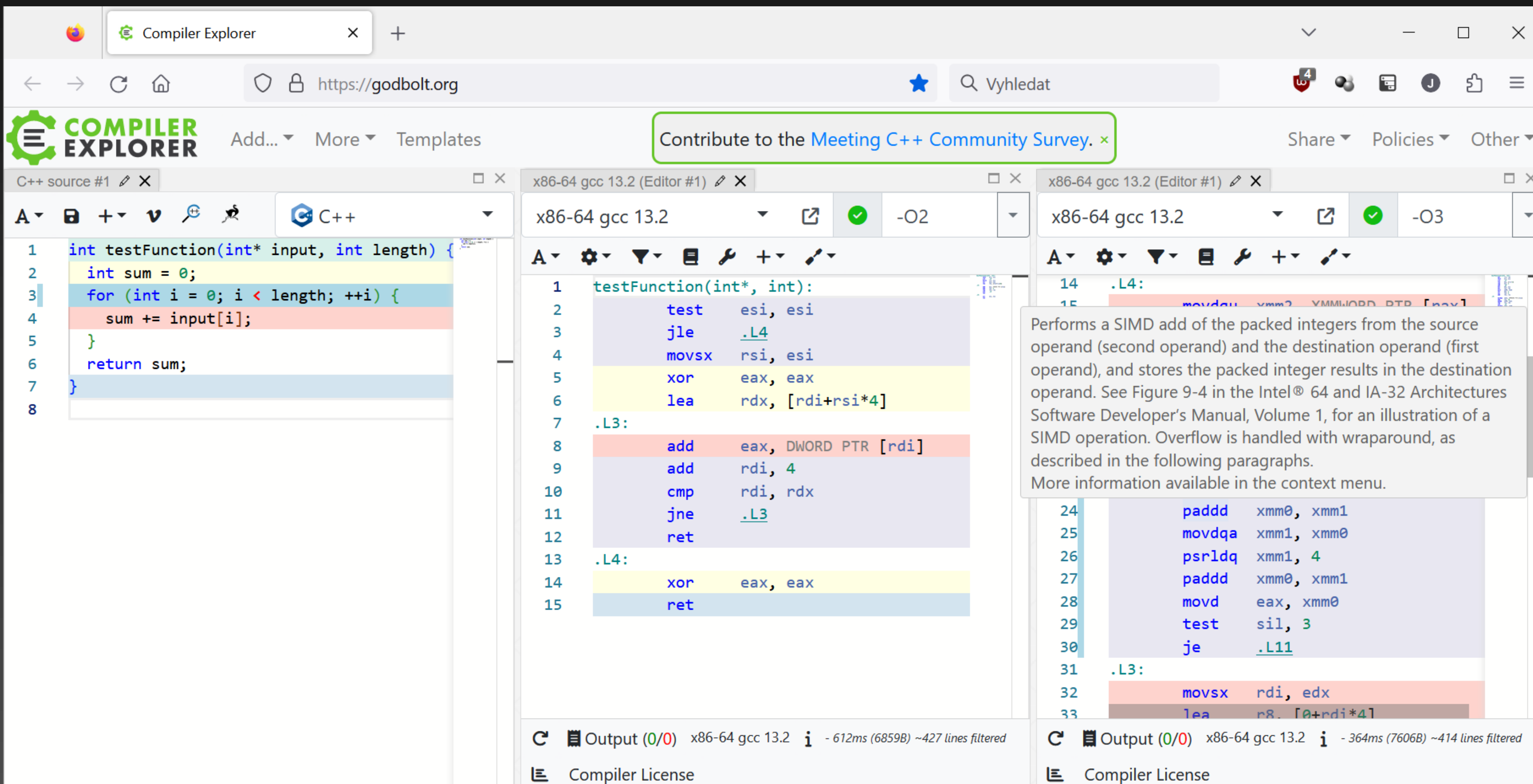
- Synchronisation (locking) necessary
- Locks
  - mutex, critical section, synchronized methods, semaphor, …

Implicit vs. Explicit Parallelism

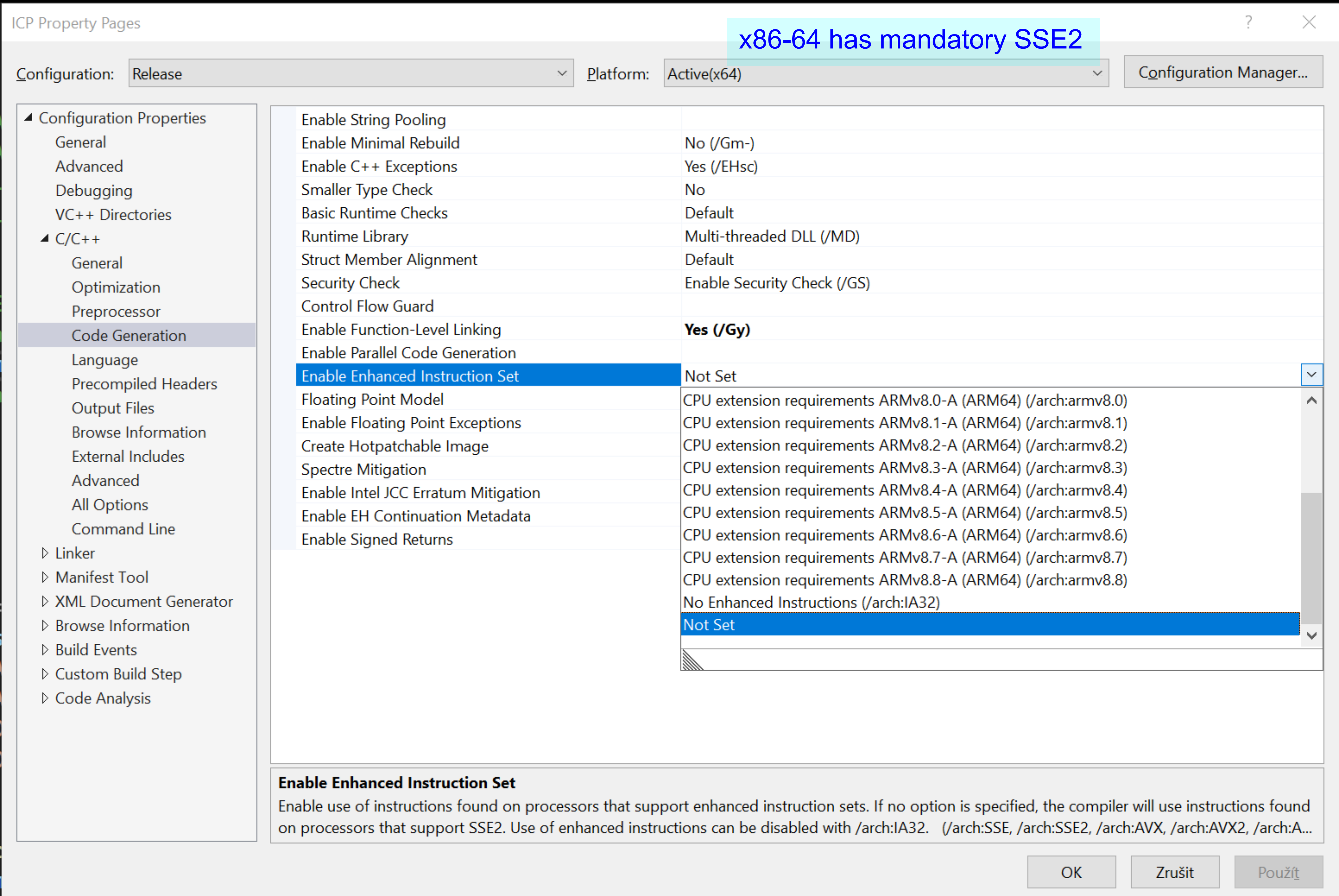
- Implicit
  - Automatic parallelisation by compiler
    - both instruction level and pieces of source code (usually for loops)
  - Precompiled libraries (OpenCV)
  - Implicitly parallel programming languages
    - LabView, Matlab (1:N)
  - No effort to splitting, comm, sync
  - Smaller control over runtime, smaller efficiency, overhead is hidden

Compiler Based Implicit Parallelism

- Modern compilers perform auto-vectorisation
  - Compiler Explorer https://godbolt.org/



x86-64 has mandatory SSE2



C++ Parallel Algorithms

- Let the compiler do all the hard work for you (min. C++ 17)
  - Add execution policy as first argument to algorithm call
- Execution policies
  - **`std::execution::seq`**
    - Run sequentially, no parallelism
  - **`std::execution::par, std::execution::par_unseq`**
    - Request to compiler to run in parallel
    - Promise by user that code is safe to run in parallel; no data races
  - **`std::execution::unseq, std::execution::par_unseq`**
    - Request to compiler to vectorize
    - Promise by user that code is safe to vectorize; no data races or locks

C++ Parallel Algorithms Examples

```
std::sort(items.begin(), items.end(), compare_by_price);
```

→

```
std::sort(std::execution::par, items.begin (), items.end(), compare_by_price);
```

```
std::fill(v.begin(), v.end(), 1);
```

→

```
std::fill(std::execution::par_unseq, v.begin(), v.end (), 1);
```

Some of C++ Parallel Algorithms

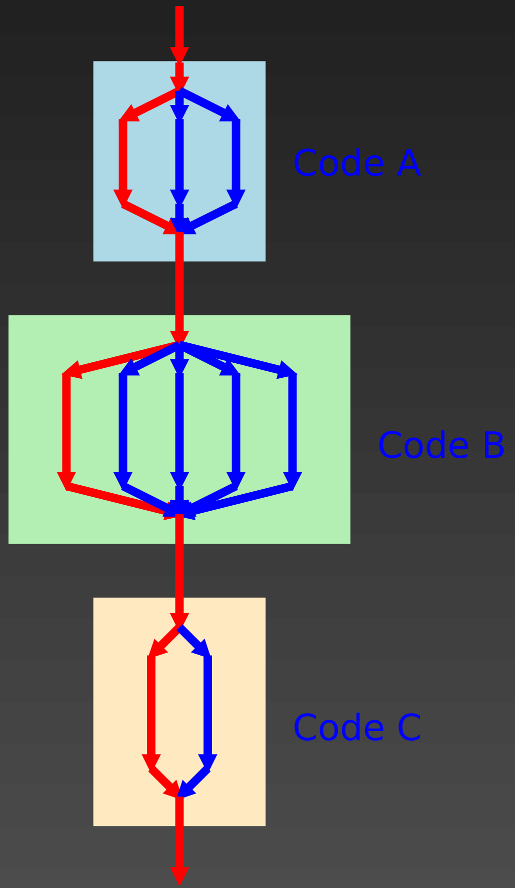
```
for_each, for_each_n, transform

find, find_if, find_end, search, count, any_of, adjacent_find

copy, copy_if, move, fill, replace, generate, rotate

sort, stable_sort, partial_sort, nth_element, is_sorted

reduce, transform_reduce, inclusive_scan, exclusive_scan
```

Implicit vs. Explicit Parallelism

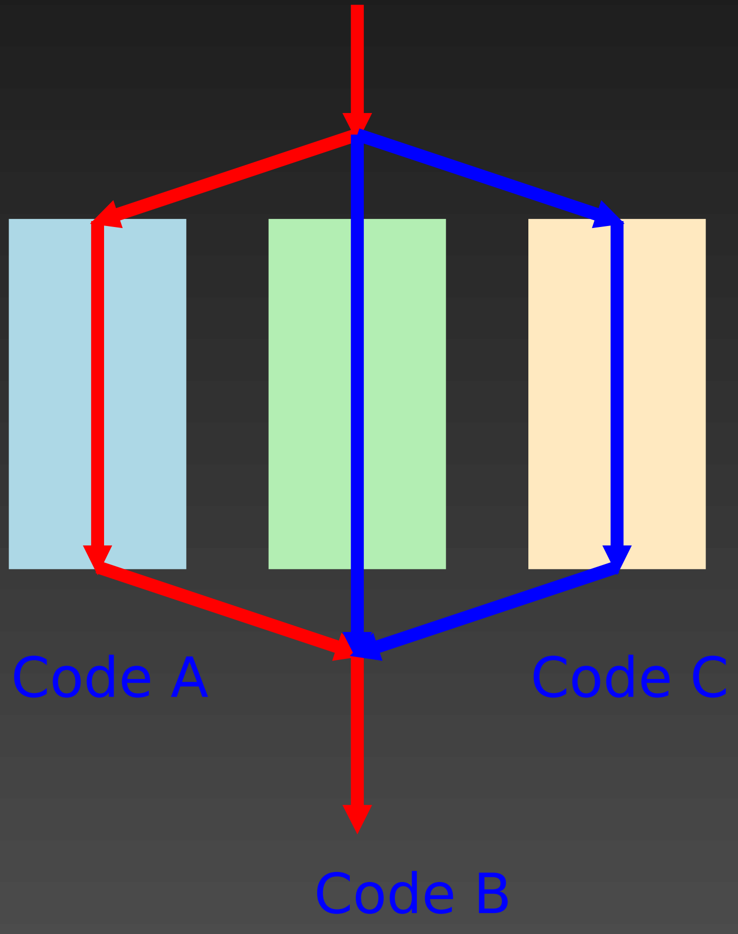
- Explicit
  - Precise control of concurrency, sync, comm using compiler directives, function calls etc.
    - overhead is visible and controllable
  - Directives
    - code splitting
    - synchronisation
    - communication
  - Full user controll, higher efficiency possible
  - thread API, OpenMP, MPI

Data vs Task Parallelism

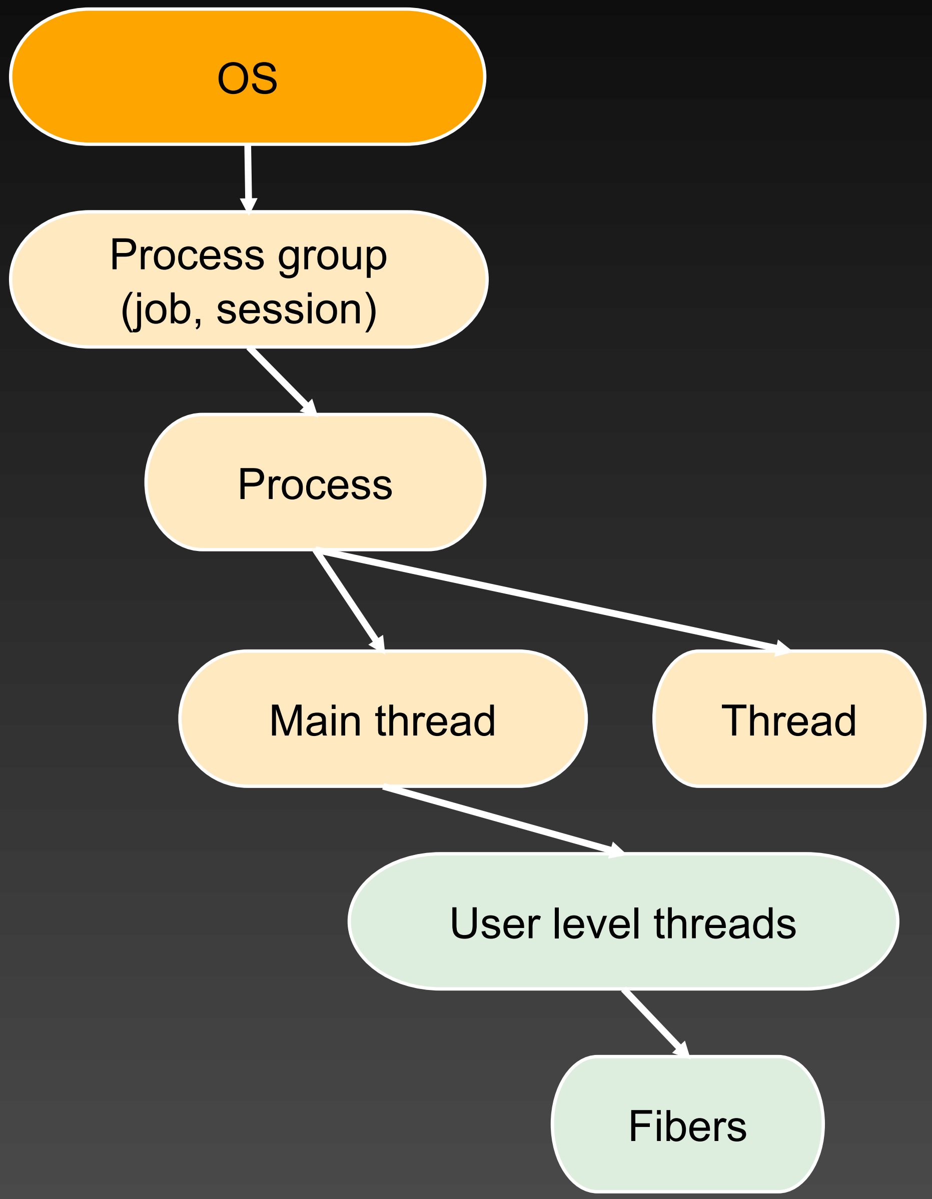
- Data-parallel
  - data are distributed
  - thread code (nearly) same



- Task-parallel
  - code is distributed

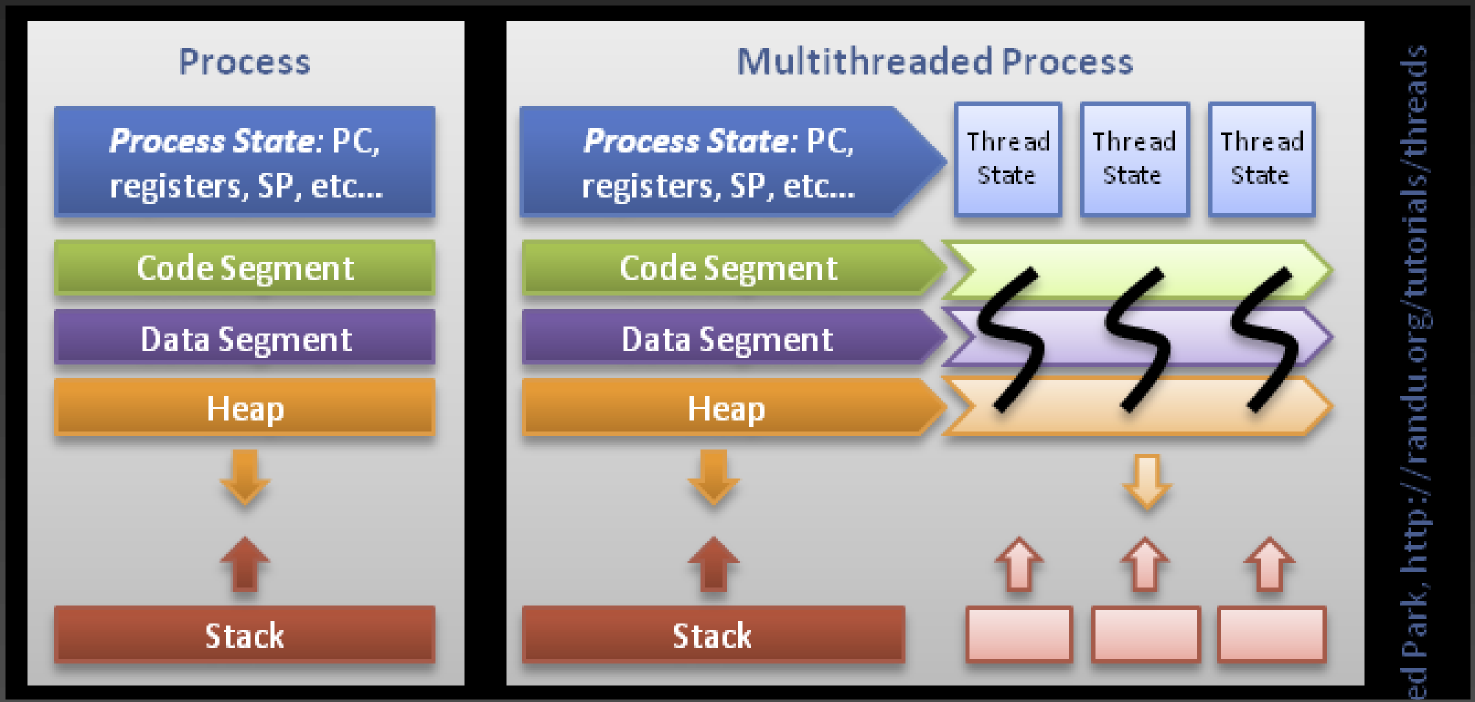


Task Hierarchy in OS



Process vs. Thread

- Process
  - Virtual memory, privileges, code, PID, priority
  - at least one thread
- Thread
  - memory is shared for all threads
  - thread local: only stack, registers, thread ID



Splitting APP to Threads

- Usually data and task parallelism
- Design patterns
  - Master/Slave + thread pool
    - master thread scatter and gather data + control others, does NOT compute itself
    - workers (slaves) threads usually created in advance
  - Equal threads
    - master also works
  - Pipeline
    - task parallel, each threads does different task
    - problems if one stage is slower

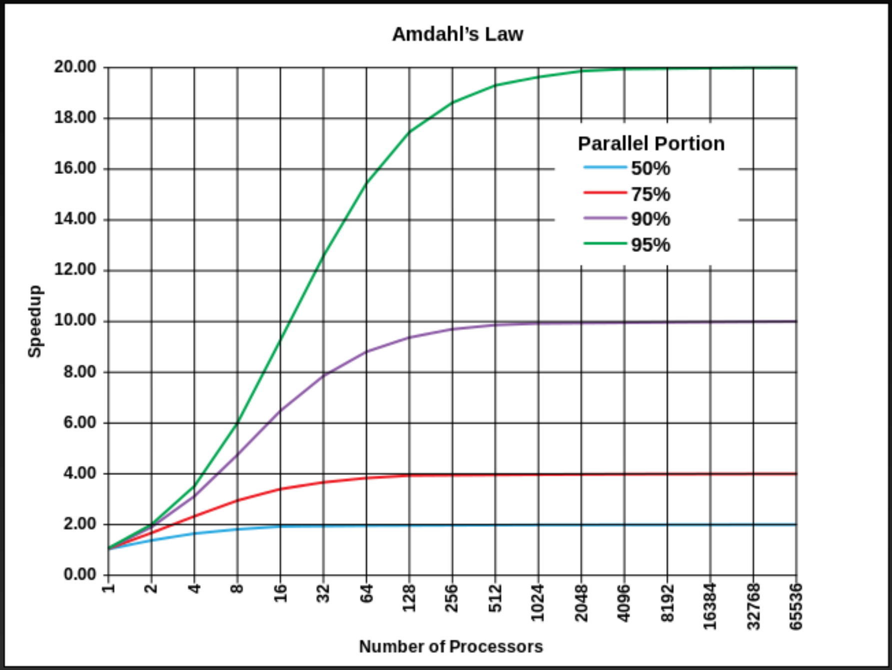
Types of Parallelism

- Fine grained
  - frequent comm and sync
  - small data blocks after shot execution
  - very latency sensitive
- Coarse grained
  - occasional communication
  - sync necessary, but not latency sensitive
- Embarrassingly parallel
  - completely independent tasks, zero comm
  - e.g. repeated run with different cmd line args

Synchronisation

- MUTEX = MUTual EXclusion
  - lock used for serialisation of thread access to resources
- Critical section
  - code between mutex locking and unlocking
  - guaranteed to be executed by only single thread at once
    - serial time
  - must be as small as possible
- Atomic operations
  - simple operations, guaranteed by hardware or library to be correct without explicit mutex (e.g. ++)

Parallel vs. Serial Time



Two independent parts **A** **B**

Original process

Make **B** 5x faster

Make **A** 2x faster

Safety

- Dangerous operation considering parallel execution
  - uncontrolled access to globals (variables and heap)
  - saving function state to global variables
  - global resource (de)allocations (files, sockets, graphics, …)
  - indirect access to data using pointers and references
  - visible side effects (modifications of volatile variables)
- Safe strategy
  - use only local variables (stack) and thread_local variables
  - code depends only on function arguments, value passing
  - all functions and subfunctions are re-entrant

Synchronisation primitives I

- Mutex – lock (locked vs. unlocked)
  - std::mutex

    void lock(); bool try_lock(); void unlock();

- Barrier (C++ 20 std::barrier, std::latch)
  - position in code, where execution of a thread is paused until all threads will arrive

- Join (fork-join)
  - gather results and exit status from all threads – join will terminate thread

Synchronisation primitives II

- Conditional variable – `std::conditional_variable`
  - call wait() for variable → thread put to sleep
  - HW watching for write into variable → wake up
  - have to check for wake-up reason

```
lock( mutex_x );
while ( not wake_me ) { sleep(cond_var, mutex_x); }
unlock( mutex_x );
```

- Semaphore – `std::counting_semaphore (C++ 20)`
  - special case: binary = mutex
  - maintains internal counter, – set to N (resources), thread enters → `acquire()` = N--
    thread exits → `release()` = N++
    on N=0 → block and wait

Threads

- Each program has one main thread, created by OS
  - Other threads created explicitly
  - Each thread can start more threads
- Thread is immediately ready to run
  - It can be started by OS scheduler before parent thread returns from create function
  - All data necessary for thread must be prepared BEFORE calling create

Thread properties

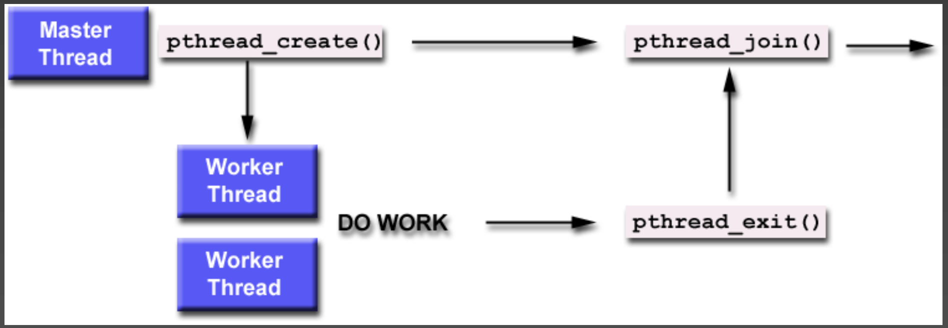
- Detached threads

  `void detach();`

  - Can not be joined with master by join()
  - Run on background, saves app resources
  - Master thread can not obtain thread exit code – no way to detect error
    - programmer must create some other way to send data (status, result) from detached thread to main thread
- `std::jthread (C++ 20)`
  - automatically rejoins on destruction
    - destructor called automatically at the end of the scope
    - no need to call join() manually

Terminating thread

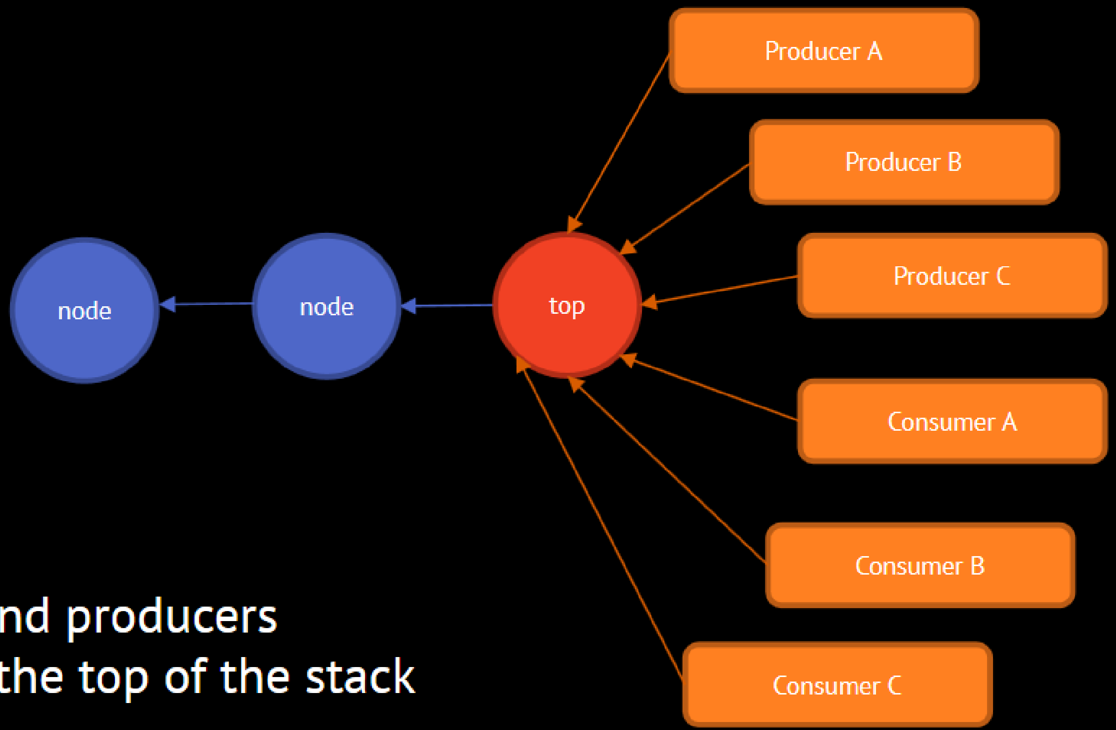
- Thread can be terminated
  - by running to the end of code
  - by ending parent thread execution by exit(), abort(), return, std::terminate() …
  - by ending master thread other than return (kill, exit, abort...)
- Thread after termination
  - process resources (fd, IPC, mutex, …) created (opened) in thread are NOT closed (deallocated) – global resources
  - heap data referenced only from thread must be released before exit – otherwise memory leak (system will release all resources on process termination, not thread termination)
- Join
  - Thread structures (status code) are cleared after thread join
  - Blocking call - wait for thread to finish
  - Join necessary if we want to know exit status code



Producer – Consumer

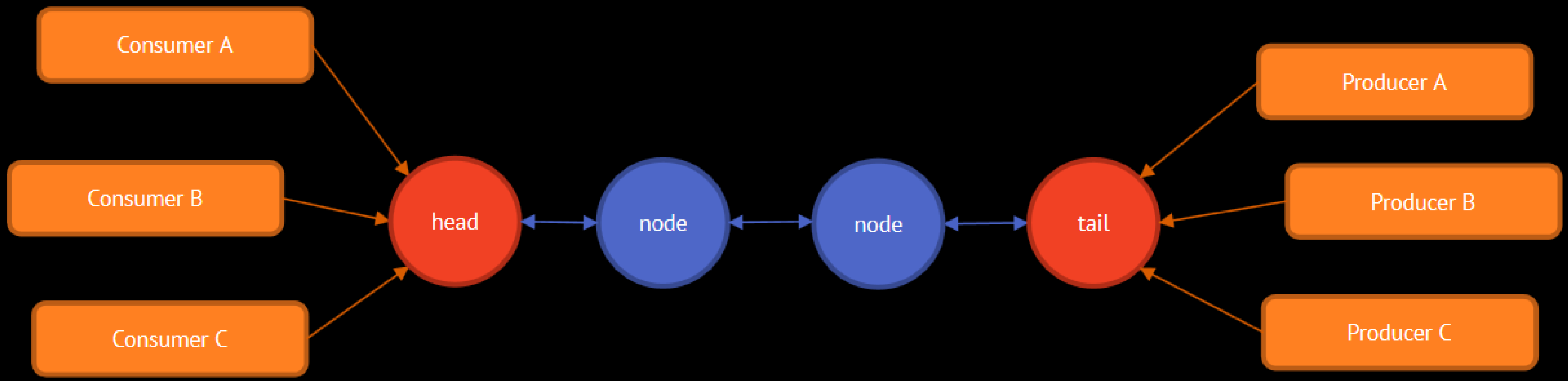
- Thread P produces data, C consumes
- Possible solution
  - shared data storage + mutex
- Better
  - common queue
  - counting semaphore, P increases, C decreases
  - at zero C can be put to sleep
  - problem: more C or P – removing and inserting is not atomic, PxC resource overwriting
- Best
  - conditional variables
  - any amount of C a P, single storage, no busy wait

Data Structures
ConcurrentStack



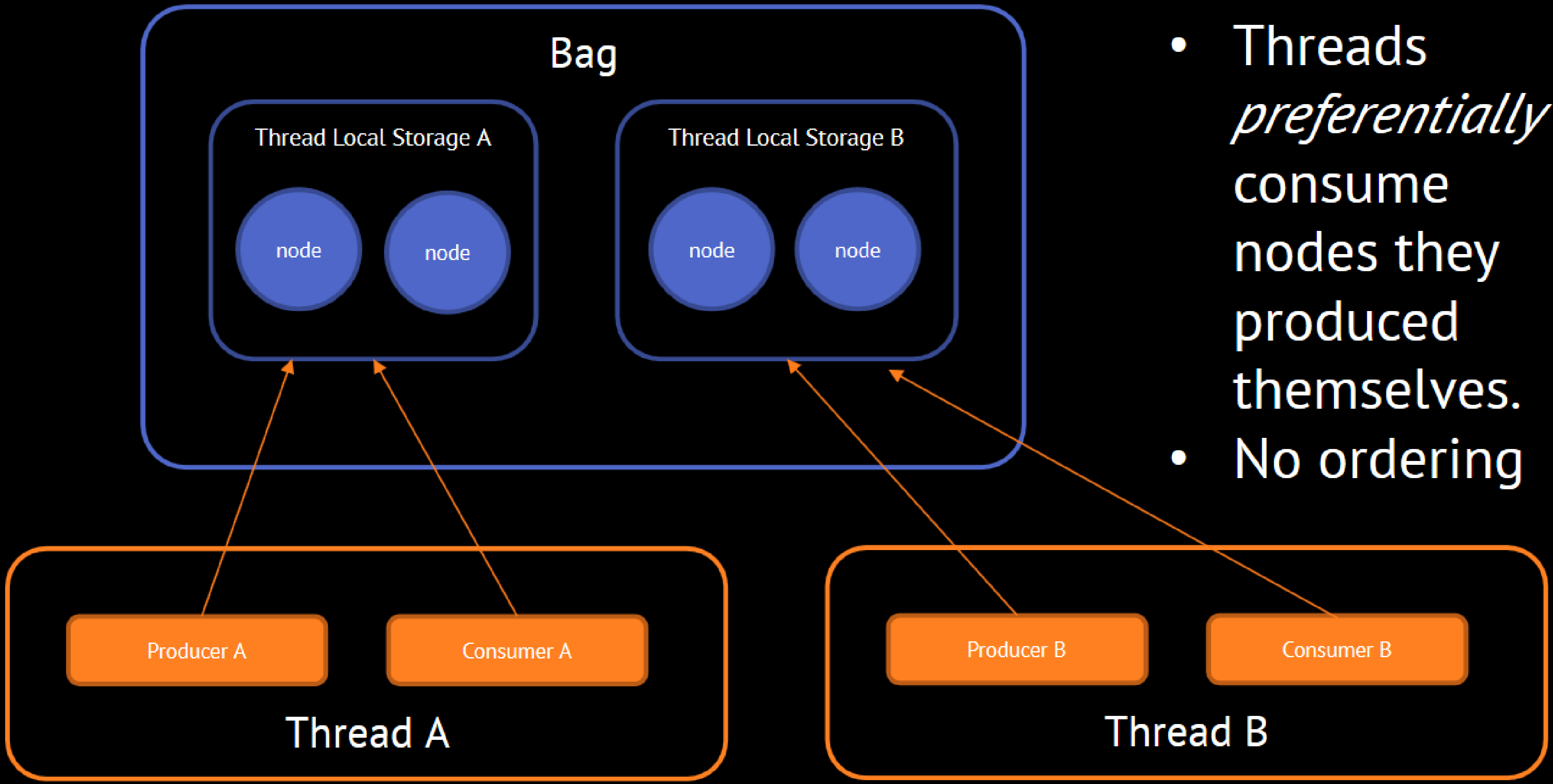
Consumers and producers
compete for the top of the stack

Data Structures
ConcurrentQueue



Consumers compete for the head.
Producers compete for the tail.
Both compete for the same node if the queue is empty.

Data Structures
ConcurrentBag



- Threads *preferentially* consume nodes they produced themselves.
- No ordering

HW Threads in C++

- header <thread>
- How many HW threads are supported by my CPU? (including SMT)

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

int main(int argc, char** argv) {
    const unsigned int hw = std::thread::hardware_concurrency();


    std::cout << "Wait a moment...\n";                   // End-Of-Line without flush (faster)

    std::this_thread::sleep_for(std::chrono::milliseconds(25));
    std::this_thread::sleep_for(std::chrono::seconds(3));

    std::cout << "Got HW threads: " << hw << std::endl;     // End-Of-Line with implicit flush (safer, slower)

    return EXIT_SUCCESS;
}
```

Synchronization by Fork & join

- create single thread vs. multiple threads, pass a parameter

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

void thread_code(void) {
    std::cout << "Hello world from thread: ID=" <<
                 std::this_thread::get_id() << "\n";

}

int main(int argc, char** argv) {
    std::thread my_thread(thread_code); // FORK
      // now thread_code() is running in separate thread...




    my_thread.join();                   // JOIN

    return EXIT_SUCCESS;
}
```

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

static const int num_threads = 10;

void thread_code(const int i) {
   std::cout << "Hello world from thread [" << i <<
                "] : ID=" << std::this_thread::get_id() <<
                "\n";
}

int main(int argc, char** argv) {
    std::vector<std::thread> threads;

    threads.resize(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        threads[i] = std::thread(thread_code, i);
      // additional parameteres passed to thread function
    }

    for (int i = 0; i < 10; ++i) {
        threads[i].join();
    }

    return EXIT_SUCCESS;
}
```

Thread Identification

- Identify main thread & execute code accordingly

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

std::thread::id main_thread_id = std::this_thread::get_id();

void am_i_main(void)
{
    if (main_thread_id == std::this_thread::get_id())
        std::cout << "This is the main thread.\n";
    else
        std::cout << "This is not the main thread.\n";
}

void thread_code(void) {
    std::cout << "Hello world from thread: " << std::this_thread::get_id() << ". ";
    am_i_main();
}

int main(int argc, char** argv) {
    std::thread my_thread(thread_code);

    am_i_main();

    my_thread.join();

    return EXIT_SUCCESS;
}
```

```
This is the main thread.
Hello world from thread: 3832. This is not the main thread.
```

std::mutex

DO NOT USE

some lock_guard variant

USE

- hardware synchronised inter-thread communication
  - in critical section, more complex operations are possible

```
#include <iostream>
#include <vector>
#include <chrono>
#include <thread>
#include <mutex> // std::mutex, std::scoped_lock

static std::mutex my_mutex;

static const int num_threads = 10;

void thread_code(const int tid, int& result) {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::thread::id this_id = std::this_thread::get_id();

    // try to move lock BELOW printing
    my_mutex.lock();
    std::cout << "I am " << tid <<
                 " with id " << this_id << std::endl;

    result += 1;
    my_mutex.unlock();
}

int main(int argc, char** argv) {
    std::vector<std::thread> threads;
    int result = 0;

    threads.resize(num_threads);
    for (int i = 0; i < num_threads; ++i) {
      threads[i] = std::thread(thread_code, i,
std::ref(result));
    }

    for (int i = 0; i < num_threads; ++i) {
        threads[i].join();
    }

    std::cout << "Result: " << result << std::endl;
    return EXIT_SUCCESS;
}
```

```
#include <iostream>
#include <vector>
#include <chrono>
#include <thread>
#include <mutex> // std::mutex, std::scoped_lock

static std::mutex my_mutex;

static const int num_threads = 10;

void thread_code(const int tid, int& result) {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::thread::id this_id = std::this_thread::get_id();
    std::scoped_lock lock(my_mutex);

    std::cout << "I am " << tid <<
                 " with id " << this_id << std::endl;

    result += 1;
}

int main(int argc, char** argv) {
    std::vector<std::thread> threads;
    int result = 0;

    threads.resize(num_threads);
    for (int i = 0; i < num_threads; ++i) {
      threads[i] = std::thread(thread_code, i,
std::ref(result));
    }

    for (int i = 0; i < num_threads; ++i) {
        threads[i].join();
    }

    std::cout << "Result: " << result << std::endl;

    return EXIT_SUCCESS;
}
```

Lock Guards

- RAII wrapper around mutexes
  - Constructor calls lock()
  - Destructor calls unlock()
- Guarantee that the mutex is always released

`std::scoped_lock` (C++ 17)

Lock Guards `std::scoped_lock` (C++ 17)

- Constructor takes one or more mutexes
  - Calls lock() on each of the mutexes
- Destructor calls unlock() on each of the mutexes
- Not copyable or movable
- No member functions or other operations

```
std::scoped_lock my_lock(mutex_a , mutex_b);
```

- `std::scoped_lock` is useful for avoiding deadlock
- If given multiple mutexes, always locks them in the same order

Lock Guards
`std::unique_lock`

- Owns a mutex
- Destructor calls unlock() on the mutex if the mutex is locked
- Has lock(), unlock(), and several other member functions
- Movable, but not copyable
- Useful when you need more control over when the mutex is locked

Atomic

- hardware synchronised inter-thread communication
  - use for smallest possible amounts of data (communication time is serial time)

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

static const int num_threads = 10;

void thread_code(const int tid, std::atomic<int>& result) {
    std::cout << tid << std::endl;

    result += 1;
}

int main(int argc, char** argv) {
    std::vector<std::thread> threads;
    std::atomic<int> result(0);

    threads.resize(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        threads[i] = std::thread(thread_code, i, std::ref(result));
    }

    for (int i = 0; i < 10; ++i) {
        threads[i].join();
    }

    std::cout << "Result: " << result << std::endl;

    return EXIT_SUCCESS;
}
```

std::conditional_variable

allows sleep and wake-up of threads

```cpp
#include <iostream>,#include <string>,#include <thread>,#include
<mutex>,#include <condition_variable>

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
    // Wait until main() sends data
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, [] {return ready; });

    // after the wait, we own the lock.
    std::cout << "Worker thread is processing data\n";
    data += " after processing";

    // Send data back to main()
    processed = true;
    std::cout << "Worker thr. signals data processing completed\n";

    // Manual unlocking is done before notifying, to avoid
    // waking up the waiting thread only to block again (see
    // notify_one for details)
    lk.unlock();
    cv.notify_one();
}
```

```cpp
int main()
{
    std::thread worker(worker_thread);

    data = "Example data";
    // send data to the worker thread
    {
       std::scoped_lock<std::mutex> lk(m);
       ready = true;
       std::cout << "main() signals data ready for processing\n";
    }
    cv.notify_one();

    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, [] {return processed; });
    }
    std::cout << "Back in main(), data = " << data << '\n';

    worker.join();
}
```

Thread-Safe Version of Container

- add lock to enforce exclusive access

```
#include <deque>
#include <mutex>                  // std::mutex,std::scoped_lock


template<typename T>
class synced_variable {
protected:
    std::mutex mux;
    T my_variable();

public:
    synced_variable() = default;
    synced_variable(const T&) = delete; //no copy-constructor

   // Guarded read
    const T& get() {
        std::scoped_lock lock(mux);
        return my_variable;
    }

    // Guarded write
    void set(const T& value) {
        std::scoped_lock lock(mux);
        my_variable = value;
    }

 }
```

Thread-Safe Version of Container

- add lock to enforce exclusive access, and condition_variable to avoid busy-wait

```cpp
#include <deque>
#include <mutex>              // std::mutex,std::scoped_lock
#include <condition_variable> // std::condition_variable

template<typename T>
class synced_deque {
protected:
    std::mutex mux;
    std::deque<T> de_queue;
    std::condition_variable cv_sleep;
    std::mutex mux_sleep;

public:
    synced_deque() = default;
    synced_deque(const synced_deque<T>&) = delete;
    virtual ~synced_deque() {
        clear();
    }

   // Returns and maintains item at front of Queue
    const T& front() {
        std::scoped_lock lock(mux);
        return de_queue.front();
    }

    // Removes and returns item from front of Queue
    T pop_front() {
        std::scoped_lock lock(mux);
        auto t = std::move(de_queue.front());
        de_queue.pop_front();
        return t;
    }
```

```cpp
    // Adds an item to back of Queue
     void push_back(const T& item) {
         std::scoped_lock lock(mux);
         de_queue.emplace_back(std::move(item));

         std::unique_lock<std::mutex> ul(mux_sleep);
         cv_sleep.notify_one();
     }

    // Returns true if Queue has no items
     bool empty() {
         std::scoped_lock lock(mux);
         return de_queue.empty();
     }

    void wait() {
         while (empty()) {
             std::unique_lock<std::mutex> ul(mux_sleep);
             cv_sleep.wait(ul);
         }

    // const T& back();
    //       T pop_back()
    //    void push_front()
    // const T& at();
    //   size_t size();
    //          ...

     }
```

Thread Outside Class vs. Inside Class

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

void thread_code()
{

}

int main()
{
    std::thread worker(thread_code);

    worker.join();
}
```

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

class App {
public:
    App(); // default constructor
    void init(void);
    int run(void);
    void thread_code(void);
    ~App(); //default destructor
private:
};

void App::thread_code()
{
}

int App::run(void)
{
    // pass a pointer to current instance
    std::thread worker(&App::thread_code, this);
    …
    worker.join();
}


int main()
{
    App app; // create instance

    return app.run();
}
```

Generative AI Example (fail)

Please create a simple C++ program with two threads: one thread is reading images using OpenCV and then send image to second thread. Second thread displays the image using OpenCV.

```
#include <iostream>
#include <thread>
#include <opencv2/opencv.hpp>

using namespace cv;

void readImage(Mat& image) {
	VideoCapture cap(0);
	while (true) {
		cap >> image;
		if (image.empty()) {
			std::cerr << "Error: no frame captured"
<< std::endl;
			break;
		}
	}
}

void displayImage(Mat& image) {
	while (true) {
		imshow("Display window", image);
		if (waitKey(1) >= 0) {
			break;
		}
	}
}

int main() {
	Mat image;
	std::thread t1(readImage, std::ref(image));
	std::thread t2(displayImage, std::ref(image));
	t1.join();
	t2.join();
	return 0;
}
```

Can you add synchronization between threads?

```
using namespace cv;

std::mutex mtx;

void readImage(Mat& image) {
	VideoCapture cap(0);
	while (true) {
		mtx.lock();
		cap >> image;
		if (image.empty()) {
			std::cerr << "Error: no frame captured"
<< std::endl;
			break;
		}
		mtx.unlock();
	}
}

void displayImage(Mat& image) {
	while (true) {
		mtx.lock();
		imshow("Display window", image);
		if (waitKey(1) >= 0) {
			break;
		}
		mtx.unlock();
	}
}

int main() {
	Mat image;
	std::thread t1(readImage, std::ref(image));
	std::thread t2(displayImage, std::ref(image));
	t1.join();
	t2.join();
	return 0;
}
```

Conclusion

- Concurrency is hard
- Use parallel algorithms when appropriate
- Avoid data races at all costs
  - Share less data
  - Lock guards (and mutexes)