

## Lectures

### L1: Introduction to Concurrency

#### Concurrency

- Concurrency is pervasive when modern computers have several cores and types of memory
- $\geq 2$  activities making progress at the same time (overlapping time periods)
- Involves interleaving of instructions from different activities

#### Parallelism

- $\geq 2$  processes executing and making progress *simultaneously*
- Hardware dependent: requires hyperthreading (SMT), or multi-core and hardware threads

#### Processes Vs threads

- Independent Vs Shared memory (address) space
- Both use independent stack
- Expensive Vs Cheap context switch
- OS facilitated vs Non OS facilitated inter-process/thread Communication
- Expensive (copy on write mediates this somewhat) Vs Cheap creation

#### Interrupts

- Asynchronous (independent to program execution)
- Used by OS to interact with the programme
- Triggered by external events (e.g. I/O, timer, hardware failure)

#### Exceptions

- Synchronous (dependent on program execution)
- Used by process to interact with the OS
- Triggered by process error (e.g. underflow, overflow)

#### User thread

- Library created, linked to one kernel thread

#### Race condition

- Outcome depends on relative ordering of operations on *ge* 2 Threads
- a flaw that occurs when the timing or ordering of events affects a program's correctness

#### Data Race

1.  $\geq 2$  concurrent threads concurrently access a shared resource without Synchronisation / fixed ordering
2. At least one modifies shared resource
3. Causes undefined behaviour

#### Mutex

- Creates critical section can be treated as a large atomic blocks
- Only one thread at a time
- Supported by a hardware instruction (CAS, test and set etc)
- **Properties:** Mutex, progress, bounded wait, performance
- Provides **serialisation** (less concurrency)

#### Critical section

- Safety: nothing bad happens
- Liveness: Something good (progress) happens
- Performance: depends on aggregate performance of all threads

#### Locks

- Primitive that is provided by the hardware, minimal semantic
- E.g. Test and set

#### Deadlock iff

1. Mutex: One resource held in a non-shareable state
2. Hold and wait: One process holding one resource and waiting for another resource
3. No-preemption: Resource and critical section cannot be aborted externally
4. Circular wait
5. Note: Lock free can deadlock

#### Dealing with deadlock

- Prevention: Eliminate one of the above conditions (E.g. hold all locks at the start)
- Detection and recovery: Look for cycles in dependencies (E.g. wait for graph)
- Avoidance: Control allocation of resources

#### Starvation

- One process cannot progress because another process is holding on a resource it needs
- Side effect of scheduling algorithm
- Wait-die and wound-wait are possible solutions, if priority of processes is preserved

#### Advantages of concurrency

- Performance
- Separation of concerns

#### Disadvantages of concurrency

- Maintenance and debugging

#### Task parallelism

1. Do the **same type of** work faster
2. Task dependency graph can be parallel
3. Make tasks specialists: Same type of tasks are assigned to the same thread
4. Divide a sequence of tasks among threads to solve complexed task
5. **Pipeline:** 1 type of thread for one phase of execution

#### Data parallelism

1. Do **more work** in the same amount of time
2. Divide data to chunks and execute by different threads
3. Embarrassingly parallel tasks

#### Challenges of concurrency

1. Finding enough parallelism: Amadahl's law
2. Granularity of tasks
3. Locality
4. Coordination and Synchronisation
5. debugging
6. Performance and monitoring

## L2: Tasks, threads, synchronisation in modern C++

#### History of C++

- 1998: No support for multithreading
  1. Effects of language model are assumed to be sequential and there are no established memory model
  2. Different libraries used different memory models
  3. Execution threads were not acknowledged
- 2011: C++11
  1. Standard threads are implemented

2. Thread aware memory model. Do not rely on platform specific extensions to guarantee behaviour
3. Atomic operations library, class to manage threads, protected shared data etc.

#### Four ways to manage threads

1. Declare a function that returns a thread

```
void hello() {
    std::cout << "Hello_Concurrent_
    World\n";
}

int main() {
    std::thread t(hello);
    t.join(); // existing thread waits for
              t to finish
}
```

2. Thread with a function object

```
class background_task {
public:
    void operator()() const {
        do_something();
        do_something_else();
    }
};

/* Callable object */
background_task f;
std::thread my_thread(f);
```

- `std::my_thread(background_task())` declares a function that takes a single parameter (type `*f()`  $\rightarrow$  object)
  - This is not the same as using a function object!
3. Threads with a lambda expression (local fn instead of a callable object)

```
std::thread my_thread([]{
    do_something();
    do_something_else();
});
```

#### Wait

- Uses `join()` on the thread instance exactly once
- Use `joinable` to check
- Local variables do not go out of scope
- Blocking

#### Detach()

- Local variable passed might go out of scope and 'disappear' during runtime, causing invalid access for the detached thread
- Example

```
void oops() {
    int local_state = 0;
    /* Reference passed might become
       invalid */
    func my_func(local_state);
    std::thread my_thread(my_func);
    my_thread.detach();
} /*oops ends here and local_state will
   be destroyed */
```

- Not blocking

#### Passing arguments

1. by value `std::thread(f, 3, "hello")`
2. by reference `std::thread(f, 3, buffer)`
  - Buffer is a charbuffer that only gets converted to str when we call f
  - Hence it is possible for buffer to go out of scope
  - Fix: Use explicit cast `std::thread(f, 3, std::string(buffer))`
  - **Major issue** with passing by reference is that threads outside of the scope can use it in **unsafe** ways. E.g. Not using mutex on shared data, deletion etc
3. by copy

```
void update_data_for_widget(widget_id w,
    widget_data& data);

void oops_again(widget_id w) {
    widget_data data;
    /* a copy of data is passed */
    std::thread t(update_data_for_widget,
        w, data);
    display_status();
    t.join();
    /* changes made to the copy is not
       reflected to other threads */
    process_widget_data(data);
}
```

- Fix: use reference `std::threadt(update_data_for_w, w, std::ref(data))`

#### Ownership in C++

- Owner is an object containing a pointer to an object allocated by *new* for which the owner is responsible for deleting
- Every object on free store (heap, dynamic store) must have **exactly one** owner

#### C++ Resource Management

- For scoped objects, destructor is implicit at scope exit
- Free store objects (created using *new*) requires explicit delete

#### RAII

- Binds the lifetime of a resource that must be acquired before use to the lifetime of an object

```
/* Handle interrupts using RAII */
void enqueue(Job job) {
    std::unique_lock lock{mut}; //
    constructor locks mutex
    jobs.push(job); // destructor unlocks
    mutex
}
```

#### Lifetime

- Lifetime begins when storage is obtained and its initialization is complete (except `std::allocator::allocate`)
- Lifetime ends when :
  - Non-class type (int): destroyed
  - Class type: When destructor is called
  - Reference: begins with initialisation and ends when destroyed. A dangling reference is possible.

#### Ownership of thread

- Moveable but not copyable

```
void some_function();
void some_other_function();
std::thread t1(some_function);
/* t1 no longer references the thread */
std::thread t2 = std::move(t1);
/* t1 now owns a new thread */
t1 = std::thread(some_other_function);
std::thread t3;
/* t3 owns the thread running some
function */
t3 = std::move(t2);
/* t1 already owns a thread, this will
trigger a runtime error */
t1 = std::move(t3);
```

- C++ compiler cannot catch this
- Ownership can be moved out of a function and moved into another function

```
/* Transferring out of a function */
std::thread g() {
    void some_function();
    std::thread t(some_function);
    return t; // ownership transferred
              out of g()
}
```

```
/* Transferring into a function */
void f(std::thread t);
void g() {
    void some_other_function();
    std::thread t(some_other_function);
    f(std::move(t)); // ownership
                     transferred into f()
}
```

### Mutex in C++

- `std::lock_guard` locks the mutex upon initialisation, unlocks upon destruction
- `std::lock_guard < std::mutex > some_mutex;`
- Group mutex and protected data together in a class rather than use global variables
- Never pass data or pointers (via returns, storing in externally visible memory, as input to functions etc) when their usage is not guaranteed to be safe

### Types of lock guards

- **lock guard** no manual lock, can lock many or one mutex at once without deadlock
- **Scoped lock** accepts and locks a list of mutexes. Can be unintentionally initialized without a mutex
- **unique lock** manual unlock, defers locking using `std::deferlock`, only single mutex.

### Condition Variable

- Use condition variables to wait for an event to be triggered by another thread
- Avoids busy waiting

```
std::condition_variable.wait(lock,
    []{ return predicate; });
```

- if condition is satisfied, returns

- unlocks the mutex and places the thread in block state if condition is not satisfied
- `std::condition_variable.notify_one()`; to notify one thread waiting on the cond

### Spurious Wake

- Thread wakes up from waiting, but is blocked again as the resource required is not available
- Leads to unnecessary context switching
- Use conditionals to prevent spurious wake

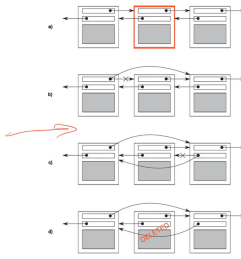
### Shared DS - Invariants

- Often broken during an update

#### Case study: Doubly LL

\*Invariant is broken during the delete

- Identify the node to delete: N.
- Update the link from the node prior to N to point to the node after N.
- Update the link from the node after N to point to the node prior to N.
- Delete node N.



- invariant is temporarily broken during an update, and we need to prevent objects from accessing the DS during this time

## L3: Atomics and Memory Model in C++

### Reordering of operations

- Compiler may reorder (potentially conflicting) actions for performance
- Not visible to programmers

### As-if rule

- Reordering is allowed as long as:
  - At termination, data written to files is exactly as if the program was executed as written (same final state)
  - Prompting text that is sent to interactive devices will be shown before the program waits for input
  - Programs with undefined behaviour is exempted from these rules.

### Multi-threading aware memory model

- Using synchronisation constructs (mutexes, barriers etc) should preclude the need for a memory model since they serialise threads
- Memory model gives us more flexibility and speed by getting us closer to the machine

### Structure of memory model

- Every object has a memory location, some occupy exactly one, some occupy many
- The changes in memory location / what is stored there affects other threads

### Modification Order

- Compose of all writes to an object from all threads in the program
- MO varies between runs, each object has their own MO
- The programmer is responsible that threads agree on the MO (if not, race condition happens)

### MO - Requirements

- The MO of each object is monotonic within a thread
- But the relative ordering of MO of different objects is not guaranteed

## MO - Building Blocks

### Sequenced-before (SB)

- Each line of code in a thread is sequenced before the next line
- There is **NO** sequenced before in a statement with many function calls

### Synchronises-with (SW)

- Established by a *load* from  $T_i$  reading  $T_j$ 's *store*
- Both  $T_i$  and  $T_j$  are synced with respect to the common value in the MO

- **Happens-before (HB)** When an operation happens before another operation due to SW or SB

### Interthread Happens Before (IHB)

- When a *store* in  $T_i$  established a sequenced before a *load* in  $T_j$ , *store<sub>i</sub>* happens before *load<sub>j</sub>*
- $IHB \subseteq HB$

### Visible Side effects

- Side effect of write A on O is visible to a read B on O if:
  - A HB B
  - There is no other side effects to O that happens between A and B
- If the side effect of A is visible to B then the longest contiguous subset of the side-effects to O (that B does not HB) is known as the visible sequence of side effects
- Do not think of ordering, think in terms of side effects that are visible

### Modification Order

#### MO - Seq Const

- The default
- All threads must see the same ordering of operation
- Synchronises with a sequentially consistent load of the same variable that reads the value loaded
- Does not apply to atomic operations with relaxed ordering
- Performance penalty when working with weakly ordered machine instructions (common)
- Essentially a serialised monoversion - global total order enforced
- Only guaranteed for data-race free programs (which is difficult since C++ is not as safe as Rust)

### MO - Relaxed

- Atomic operations don't conform with SW relationships
- Happens before still applies within the thread → monotonicity and SB within the thread is preserved
- No HB between load and store, different store operations from T1 can be viewed out of order by reads in T2
- T1:  $x = 1, y = 0$ . T2 can see  $y=0$  without seeing  $x=1$  since there is no SW between the two threads even though  $x=1$  HB  $y=1$  in T1.

### MO - Acquire Release

- No total modification order, but there is a partial order
- Read - acquire updates about the memory order, load - release updates about the memory order
- A link between acquire and release acts like a barrier

### MO - Mixing Models

- Seq const and Release Acquire: load and store of seq const behaves similar to release acquire
- any MO and relaxed: Relaxed behaves like relaxed but is bounded by the other more limiting MO

```
// T1
x.store(true, std::memory_order_relaxed);
```

```
y.store(true, std::memory_order_release);
// T2
while (!y.load(std::memory_order_acquire));
/* Never fires because acquire and release */
/* x.store HB y.store & y.store SW y.load */
assert(x.load(std::memory_order_relaxed));
```

## Atomic Operations

- Compiler ensures necessary synchronisation is in place and enforces MO
- Atomic ops are indivisible
- Atomic load loads either the initial value or the value stored by one of the modifications (cannot be half-done)
- Can be lock free or be implemented using mutex (which wipes off performance gains)
- Not necessarily race free

## L4: Testing and debugging in C++

### Concurrency related bugs

- Unwanted blocking: Deadlock, livelock, blocking while waiting on I/O
- Race conditions:
  - Data races: Undefined behaviour due to unsynchronised access to a shared memory location. Observable.
  - Broken Invariants:
    - Dangling pointers: another thread deleted the data being accessed
    - Random memory corruption: Inconsistent values being read due to partial updates
    - Double free: Two threads pop the same value from a queue / deleting the same memory address twice - causes possible memory leak
    - Use After Free - Reading or writing after a memory has been freed
    - Uninitialised variables - Reading from a variable that has not been initialised
  - Lifetime issues:
    - Thread outlives data
    - Call to join skipped due to an exception thrown

### Techniques to locate concurrency bugs

- Look at the code
- Testing: Difficult to reproduce, Tests do not always fail (Heisenbug)

### Guidelines for testing

- Run the smallest amount of code that could potentially demonstrate the bug to locate faulty code
- Do single threaded tests to verify the bug is concurrency related
- Run on single core system to identify issues with interleavings

### Test environment

- Number of threads: More threads increases the chance of deadlock (at least 2), contention (blocking while contending for shared resources, degrades performance), overhead
- Architecture
- Number of cores
- Having memory fences and barriers to sync threads

### Designs for testability

- Responsibility for each function and thread should be clear

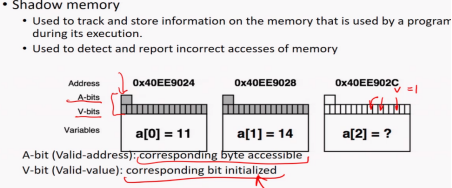
- Check that library calls are thread safe (e.g. do they use internal states to ensure correctness?)

### Techniques for testing

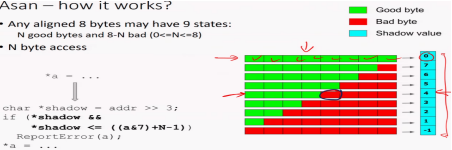
1. Stress testing
2. Use special implementation of synchronisation primitives (e.g. log when mutexes are locked, unlocked)
3. **Scalability** does the speedup scale when the number of threads increase? Contention to

### Debugging Tools

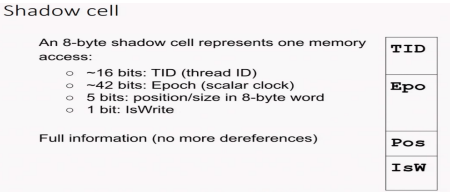
- Identify bugs:
  1. Valgrind (dynamic instrumentation)
    - Shadow memory: track and store information on the memory that is used by a program during its execution. A bits and V bits must match for valid.
    - 20x slowdown
    - Shadow memory
      - Used to track and store information on the memory that is used by a program during its execution.
      - Used to detect and report incorrect accesses of memory



2. Helgrind (dynamic instrumentation)
  - 100x slow down
  - intercepts function calls to functions and instruments
  - Detects: misuses of POSIX threads API, potential deadlocks (checking cyclic lock acquisition), data races (checks existence of HB between memory accesses)
3. Sanitizers (compilation-based approach)
  - 5-10x overhead (lmao just use rust)
  - `-fsanitize = address` or `-fsanitize = thread` or `-fsanitize = memory` (to catch uninitialised memory)
  - Output is verbose (unlike rust compiler)
4. Address Sanitizers (compilation-based approach)
  - 2x slowdown, 3x overhead
  - More efficient shadowing than valgrind



5. Thread Sanitizer
  - 5-10x slow down, 5-15x overhead
  - Function entry/exit and memory access are logged
  - An 8-byte shadow cell represents one memory access
  - Epoch: time of access
  - Pos: location accessed
  - If there is an overlap in Pos, we check epoch for evidence of HB. If no HB, then there might be a data race
  - Use graph-based deadlock detection



# Tutorials

## T1: Threads and Synchronisation

### Why mutexes work - standard argument

- Define a critical section that contains all accesess to the shared resource
- Argue that mutex guarantees mutual exclusivity of threads
  - removes interleaving, data race precluded

### Why mutexes work - theoretical argument

- Lock and unlock appears in a single total order
- Only one thread owns the lock at any pointer
- Unlock happens after lock, creating a synchronises with relationship between processes and **serialises the interleaving** - no concurrent access

### Monitor

- Allows us to block until a condition becomes true
- The monitor has:
  1. A mutex on the critical section
  2. A condition variable
  3. A condition to wait for

```
std::condition_variable cond;

Job dequeue() {
    std::unique_lock lock{mut};
    /* wait until there is a job */
    cond.wait(lock, [this]() { return
        !jobs.empty(); });
    Job job = jobs.front();
    jobs.pop();
    return job;
}

void enqueue(Job job) {
    {
        std::unique_lock lock{mut};
        jobs.push(job);
    }
    /* notify one thread waiting on the
       condition variable */
    cond.notify_one();
}
```

## T2: Atomics in C++

### Data races and undefined behaviour

- For undefined behaviour, the C++ compiler **will allow it to be compiled**
- Compilers are free to do any reordering and the answer cannot be predicted.
- E.g. GCC and Clang will return different runtime outputs as they compile it differently
- See snippet 1

### Mutexes Vs Atomics

- Consider the following implementation of a counter

```
int counter = 0
std::mutex m;
void t1 {
    for (int i = 0; i < 1000000; i++) {
        std::lock_guard lock{mut};
        counter++;
    }
}
```

```
    }
}

std::atomic<int> atomic_counter = 0
void t2 {
    for (int i = 0; i < 1000000; i++) {
        atomic_counter.fetch_add(1,
            std::memory_order_seq_cst);
    }
}
```

- The atomic version is much faster than the mutex version
- Mutex calls lock and unlock multiple times around the add operation, which compiles to more instructions and may cause the threads to sleep / fight for access
- The atomic version is a single instruction *lock addl \$1*, which is much faster

### Memory order and performance

- Seq cst  $\geq$  Acquire-Release  $\geq$  Relaxed
- Seq cst's store uses *xchg* (requires the processor to have exclusive access to shared mem) while the other orders allows x86 to use a simple *mov* instruction

### Forcing ordering with *std::atomic<int>*

- Compiler reorders instructions to preserve visible side effects and optimise
- We can use atomics to force the compiler to preserve ordering
- But this also forces the compiler to use the more expensive *xchg* instruction, since atomics use seq cst by default

### Memory order - atomics

- Note: data races **is not possible** when atomics are used. If load and store are not atomic, then the output can theoretically be anything (segfault, garbage value etc)

Thread 1	Thread 2
<pre>x.store(1, stdmo::relaxed); // (a) y.store(2, stdmo::release); // (b) x.store(3, stdmo::relaxed); // (c) z.store(4, stdmo::release); // (d) x.store(5, stdmo::relaxed); // (e)</pre>	<pre>while (y.load(stdmo::acquire) != 2) { } // (p) cout &lt;&lt; x.load(stdmo::relaxed); // (q) while (z.load(stdmo::acquire) != 4) { } // (r) cout &lt;&lt; x.load(stdmo::relaxed); // (s)</pre>

- The first cout can print 1,3,5. The secon cout can print 3,5. They will never print 0 due to the acquire release.

### Fences

- Enforces memory order without modifying the data
- Can be used with Relaxed MO to enforce ordering while preserving concurrency
- **Memory Barrier** puts a line that certain operations cannot pass
- **Atomic fence** with acquire release MO prevents all preceding read and writes from moving past all subsequent stores

## T3: Debugging Concurrent C++ Programs

### Protecting shared resource with unique\_ptr

- std::unique\_ptr is a smart pointer that owns and manages another object through a pointer and disposes of that object when the unique\_ptr goes out of scope.
- Destroys object with the provided *deleter*

```
{
```

```
std::unique_ptr<int> foo =
    std::make_unique<int>(5);
/* custom deleter for File */
auto deleter = [](FILE* f) { fclose(f); };
auto bar = std::unique_ptr<FILE,
    decltype(deleter)>(fopen("file.txt",
    "w"), deleter);
}
/* Foo and bar are destroyed here */
```

### Protecting against shared lifetimes

- std::shared\_ptr is a smart pointer that retains shared ownership of an object through a pointer.
- When shared\_ptr is copied, we increase the count by 1. When it is destroyed, we decrease the count by 1. When count is 0, we delete the object
- While deleting shared\_ptr is thread safe (since it just deceremnts the counter), the object wrapped by shared\_ptr may not be (data races can still happen)

**Classical Synchronisation Problems**

**Comparison between Rust, Go, C++**

**Ownership**

- C++ has RAll to manage resources, moveable but not copyable reference

**Testing and debugging**

- C++ cannot catch many race conditions and concurrency bugs at compile time, and requires testing at run time
- Tsan and Asan has verbose output (unlike rust compiler)