C++ Memory Model
from C++11 to C++23

Alex Dathskovskky

2024
About Me:

alex.datskovsky@speedata.io
www.linkedin.com/in/alexdathskovsky
https://www.cppnext.com
Important Question
Does the processor executes the program as you wrote it?
Short Version

• Usually Not
Longer Version

• It will run what you have intended but:
  • Compilers reorder and change operations
  • CPU’s use threading and OOO execution
Compiler Optimisations
If you were the compiler, what would you do?

```cpp
constexpr std::size_t N = 100;
auto data = get_int_array<N>();
std::size_t p_sum{};
for (std::size_t i{}; i < N; i++){
    auto p = get_pivot(N);
    p_sum += data[i] * data[p];
}
```
If you were the compiler, what would you do?

- Compiler detects that `get_pivot` yields the same value
- It's better to call `get_pivot` once and call the value from register again and again
- `data[p]` has the same value, it will be stored in a register
Compiler can do even better

- Modern (X86) CPUs have vectorization
Compiler can do even better

• If everything can be known in compile time
Algebraic Simplifications

\[(v + 1) + 3 \rightarrow v + (1+3) \rightarrow v + 4\]
Strength Reduction

```c
uint64_t add_and_devide_u(uint64_t a, uint64_t b){
    return (a+b)/2;
}
```

```
add_and_devide_u(unsigned long, unsigned long):
    lea     rax, [rdi + rsi]
    shr     rax
    ret
```
To INT Or To UINT

ALEX DATHSKOVSKY
Loop unrolling

• Compiler may unroll your loops

```c++
for (int i{}; i<5; i++){
    std::cout << "Hello World!\n";
}
```
Loop unrolling

- Compiler may unroll your loops

```c
mov edx, 13
mov rdi, rbx
mov rsi, r14
call std::basic_ostream<char, std::char_traits<char>> & std::_ostream_insert<char, std::char_traits<char>> >
mov edx, 13
mov rdi, rbx
mov rsi, r14
```
Dead Code Removal

• If there is unnecessary code, it will probably be removed

```c
int dead_code(int size){
    int i{}, j{};
    for (; i < size; i++){
        j++;
    }
    return i;
}
```
Dead Code Removal

• If there is unnecessary code, it will probably be removed

```c
dead_code(int):
    xor eax, eax
    test edi, edi
    cmovg eax, edi
    ret
```
inlining

• inlines function calls

```c
int add1(int a) {
    return a + 1;
}

int main(int c, char** argv){
    return add1(c);
}
```
inlining

• inlines function calls

```c
main: # @main
    lea   eax, [rdi + 1]
    ret
```
inlining recursion

• Tail recursion can be inlined

```c
int add(int a, int n){
    if (n==0){
        return a;
    }
    return add(a+1, n-1);
}

int main(int c, char** argv){
    return add(c, c+10);
}
```
inlining recursion

• Tail recursion can be inlined

```
1        add(int, int):
2          lea    eax, [rdi + rsi]
3          ret
4    main:
5          lea    eax, [2*rdi + 10]
6          ret
```
CPU Optimisations
INO Execution

Source: https://en.wikipedia.org/wiki/Central_processing_unit
High Speed Query Execution with Accelerators and C++

ALEX DATHSKOVSKY

Cppcon 2022
September 12th-16th
INO Execution

Source: https://en.wikipedia.org/wiki/Instruction_pipelining
INO Execution

- instruction fetch
- if operands available execute it if not fetch them
- The instruction is executed by the functional unit
- The functional unit writes the result back to the register or memory
INO vs OOO Execution

Source:
https://www.semanticscholar.org/paper/RISC-V-Reward:-Building-Out-of-Order-Processors-in-Zekany-Tan/f7f6d27f334604c3c85f0b8d21d2a9b4d22a983

When an instruction does not depend on prior instructions, the pipeline does not need to stall.
OOO Execution : Dynamic Scheduling

- instruction fetched
- instruction dispatched to instruction queue
- the instruction waits in the queue until its input operand are available
- if operands available instruction is allowed to leave the queue before other instructions
- the instruction is issued to a functional unit
- only if all older instructions have completed the operation the result is written to register file
Dynamic Scheduling

- **Check for structural hazards**
  - an instruction can be issued if a functional unit is available
  - an instruction stalls if no appropriate functional unit available
Dynamic Scheduling

• **Check for data hazards**
  • an instruction can be executed when its operands have been calculated or loaded from memory
  • an instruction stalls if operands are not available
Dynamic Scheduling

• instruction can be executed out of order if there is no dependency on previous instructions
• ready instructions can execute before earlier instructions that are stalled
Cache $\$ $

- Cache can accelerate reads and make writes longer and more complex.

```
<table>
<thead>
<tr>
<th>Small</th>
<th>Fast</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;500B</td>
<td>0.25ns</td>
</tr>
<tr>
<td>64KB</td>
<td>1-2ns</td>
</tr>
<tr>
<td>8MB</td>
<td>5ns</td>
</tr>
<tr>
<td>4GB</td>
<td>100ns</td>
</tr>
<tr>
<td>100GB</td>
<td>1ms</td>
</tr>
</tbody>
</table>
```

CPU registers are the highest level of memory, part of the ISA.
Reordering and C++
Reordering Types

- Data dependencies must be honored
- C++ compiler may reorder any memory access under the as-if rule
- Different processors have different reordering guaranties
AS-IF Rule

- Accesses (reads and writes) to volatile objects occur strictly according to the semantics of the expressions in which they occur. In particular, they are not reordered with respect to other volatile accesses on the same thread.
AS-IF Rule

● At program termination, data written to files is exactly as if the program was executed as written

● Prompting text which is sent to interactive devices will be shown before the program waits for input

● Executing function from external libraries
AS-IF Rule - Not Always

- programs with undefined behaviour
- copy elision, the compiler may remove calls to move and copy-constructor and destructors of temporary objects even if those calls have side effects
Multi Threading and Reordering
Reordering between two threads

Thread 1
flag1 = 1;
if (flag2 == 0){
critical section}

Thread 2
flag2 = 1;
if (flag1 == 0){
critical section}
Sequential Consistency

- Result of any execution is the same as if the operations of all the processors were executed in some sequential order
- Operations of each individual processor appear in the sequence in the order specified by the program
SQC: Do we Have It?

- SQC is very expensive
- Modern compilers do not offer it (for free)
What Does C++ and Modern CPU guarantee?

SC-DRF:

- A system guaranteeing DRF-SC must define specific instructions called *synchronizing instructions*, which provide a way to coordinate different processors (equivalently, threads).

- Programs use those instructions to create a “happens before” relationship between code running on one processor and code running on another.
Example:

Thread 1
Foo(Flag)
S(a)
Bar(Flag)

Thread 2
S(a)
Bar(Flag)
Pop Quiz: can this happen?

Thread 1  
\( x = 1 \)

Thread 2  
\( x = 2 \)

Thread 3  
\( y_1 = x \)  
\( y_2 = x \)

Thread 4  
\( y_3 = x \)  
\( y_4 = x \)

\( y_1 = 1, y_2 = 2, y_3 = 2, y_4 = 1? \)
### Pop Quiz: can this happen?

\[
y_1 = 1, y_2 = 2, y_3 = 2, y_4 = 1? \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Thread 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( x = 1 )</td>
<td>4: ( x = 2 )</td>
<td>2: ( y_1 = x )</td>
<td>3: ( y_4 = x )</td>
</tr>
<tr>
<td>5: ( y_2 = x )</td>
<td>6: ( y_3 = x )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Synchronization
ADDRESSING THE ELEPHANT IN THE ROOM
Volatile

- Volatile is not a synchronization tool
- Volatile doesn’t affect threading
- It's not an atomic value
- It doesn’t add barriers
Volatile: what is it good for?

- Tells the compiler it shouldn’t optimize the memory reads and writes order.
- Alas the CPU can still reorder
Volatile: Bad Example

```cpp
int main(){
    int y = 1;
    volatile int* x = &y;

    auto w = std::thread([x](){
        for (int i = 0; i < 50; i++){
            *x += 1;
        }
    });

    auto r = std::thread([x](){
        for (int i = 0; i < 50; i++){
            fmt::print("\{\}n", *x);
        }
    });

    r.join();
    w.join();
}
```
Volatile: Most uses deprecated with C++20

- Is += a single/atomic instruction? How about ++?
- How many reads/writes are needed for compare_exchange? What if it fails?
- foo(int volatile n) int volatile foo() are meaningless.
Volatile: Most uses deprecated with C++20

- For more information watch JF’s talk about it
Synchronization cont
Compiler Code Barriers

Prevent compiler from moving reads or writes across the barrier
Compiler Code Barriers

Prevent compiler from moving reads or writes across the barrier

Thread 1
value = very_long_calc()
asm volatile("mfence" ::: "memory")
done = true //do something

Thread 2
while (not done){}
asm volatile("mfence" ::: "memory")
Locks and Atomics

Most Locks and atomic operation will act like barriers
Understanding Memory Barriers / Memory Ordering
Acquire Semantics

- property that can only apply to operations that read from shared memory, whether they are read-modify-write operations or plain loads. The operation is then considered a read-acquire. Acquire semantics prevent memory reordering of the read-acquire with any read or write operation that follows it in program order.
Acquire Semantics

All memory operations stay below the line
Release Semantics

- property that can only apply to operations that write to shared memory, whether they are read-modify-write operations or plain stores. The operation is then considered a write-release. Release semantics prevent memory reordering of the write-release with any read or write operation that precedes it in program order.
Release Semantics

All memory operations stay above the line
Memory Barrier / Acquire Release Order

All memory operations stay between the borders

Read-Aquire

Write-Release
Sequential Consistency

- This is the default and the most strict mode. Enforcing that all operations are seen in a globally consistent order across all threads, as if they were executing in a single sequential thread. This means that any read or write operation that uses this memory order will be sequenced before or after all other operations in the program, providing a globally consistent view of memory.
Sequential Consistency

All operations will be in the same order
What's The Difference?

Thread 1 | Thread 2 | Thread 3 | Thread 4
---|---|---|---
x = sr(1) | y = sr(2) | y1 = la(x) | y3 = la(y)
y2 = la(y) | y4 = la(x) | y1 = 1 | y3 = 2
y2 = 0 | y4 = 0

sr = store_release
la = load_acquire

T3: y1 = 1 y2 = 0
T4: y3 = 2 y4 = 0
What's The Difference?

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Thread 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = \text{ss}(1)$</td>
<td>$y = \text{ss}(2)$</td>
<td>$y_1 = \text{ls}(x)$</td>
<td>$y_3 = \text{ls}(y)$</td>
</tr>
<tr>
<td>$y_2 = \text{ls}(y)$</td>
<td>$y_4 = \text{ls}(x)$</td>
<td>$T3: \ y_1 = 1 \ y_2 = 2$</td>
<td>$T4: \ y_1 = 1 \ y_2 = 2$</td>
</tr>
</tbody>
</table>

$\text{ss} = \text{store\_seq\_cst}$

$\text{ls} = \text{load\_seq\_cst}$
Relaxed Semantics

- This semantics offers the weakest guarantees among memory ordering constraints. There are no guarantees about the relative order of memory accesses as observed by different threads. This means that operations performed by one thread might not be immediately visible to other threads, and the order in which operations appear to occur might not reflect the actual order in which they were executed.
Relaxed Semantics

Relaxed

Everything is possible

Relaxed
Consume Semantics
Consume Semantics

designed to exploit data dependencies between threads to provide synchronization guarantees.
Data Dependency Ordering

- Some CPU have to emit barriers to order between memory instructions (armv7, risc v)
- Data dependencies make the order defined

```
ldr w8, [x8, :lo12:val]
add w8, w8, #1
str w8, [x9, :lo12:out]
```
Data Dependency Ordering

- Data dependencies can be on memory and chained

```
ldr    w0, [x3]
ldr    w2, [x2]
add    w0, w0, w2
add    w0, w0, 40
str    w0, [x1]
```
Consume Semantics : How does it works

- **The compiler exploit the data dependency chains**
- **Instructions have to carry-a-dependency (lucky it's defined in the standard)**
- **to summarize: one evaluation carry-a-dependency if value of the first is defined in the second evaluation as an operand**
Example

```c
a = atomic_bool_value{false};
x = 0;

Thread 1
x = 1;
a.s(true);

Thread 2
z = a.lc();
if (z) y = x
```
Good Example

```c
a = atomic_int*_value{null}

Thread 1
x = 1
a.sr(&x)

Thread 2
z = a.lc()
if (z) y = *z
```
Consume Semantics

Good news consume is almost like acquire, acquire promises same things and its stronger.

Most compilers do not support consume and will emit release-acquire
What may happen here?

A1.store(4, release)
Y = A2.load(acquire)
What may happen here?

\[ Y = \text{A2.load(acquire)} \]
\[ \text{A1.store(4, release)} \]
Why is it possible?

A1. store(4, release)

Y = A2.load(acquire)
What may happen here?

\[ Y = A2.load(acquire) \]

\[ A1.store(4, \text{release}) \]
C++ Concurrency tools
std::thread

- Cross platform thread class

```cpp
std::thread([](){ std::this_thread::sleep_for(1s); }).join();
```
thread_local

- storage of the variable is defined by the thread
- helps with data races, each thread has its own copy

```cpp
thread_local uint64_t inc = 0;

void do_count(std::string view_name, int count, uint64_t & ret){
    for
    } 
    ret a: 10
    ret b: 20
}

int main
    uint64_t inc, ra, rb;
    std::lock_guard<std::mutex> a_guard(a); 
    std::lock_guard<std::mutex> b_guard(b);
    a_guard.join(); b_guard.join();
    fmt::print("ret a: {}\nret b: {}\nmain inc: {}\n", ra, rb, inc);
```
thread_local

- Please don't abuse it

```cpp
thread_local uint64_t inc = 0;

void do_count(std::string_view name, int count,
              uint64_t& ret, uint64_t* val){
    ret a: 30
    ret b: 20
    main inc: 30
}

int main(uir stc c);
std::thread b(do_count, "A", 20, std::ref(rb), &inc);
a.join(); b.join();
fmt::print("ret a: {}\nret b: {}\nmain inc: {}\n", ra, rb, inc);
```
thread_local

● Please don't abuse it
std::atomic

- Provides a portable way to perform low level atomic operations
- No torn reads and no torn writes
- Provides read-acquire, write-release and full memory barriers
Atomic Memory Order

- memory_order_relaxed – there are no sync or ordering constraints, only this operation is guaranteed to be atomic
- memory_order_acquire
- memory_order_release
- memory_order_seq_cst – this will give us sequential consistency, and this is the default mode
Atomic Memory Fence

**Int** \( a = 0 \)

**Thread 1**
\[
a = 42;\]
\[
\text{atomic\_thread\_fence(memory\_order\_release)}\]

**Thread 2**
\[
\text{atomic\_thread\_fence(memory\_order\_acquire)}\]
\[
\text{int } r2 = a \ ? \ a : -1;\]
std::atomic Without a Fence

Int a = 0
std::atomic<int> ready(0)

Thread 1
a = 42;
ready.store(1, memory_order_release)

Thread 2
int r1 = ready.load(memory_order_acquire)
int r2 = a ? a : -1;
std::atomic: Default

Int a = 0
std::atomic<int> ready(0)

Thread 1

a = 42;
ready.store(1)

Thread 2

int r1 = ready
int r2 = a ? a : -1;
std::atomic Performance: Loads

- On X86 atomic loads are just loads (primitive types)
- Can be more expansive and cause locking on other systems
std::atomic Performance: Stores

- On X86 atomic stores use xchg, and it is a full barrier
void spin_bool_foo(std::atomic<bool>& flag) {
    while (flag.exchange(true)) {
        //spin;
    }
    fmt::print("Worker Spin lock opened doing stuff \n");
    //doing stuff
    fmt::print("Worker releasing lock \n");
    flag = false;
}
std::atomic Spinlock Example Cont

```cpp
int main()
{
    std::atomic<bool> flag{true};
    auto holder = std::thread([&flag](){
        std::this_thread::sleep_for(1s);
        fmt::print("Holder releasing lock \n");
        flag = false;});

    auto worker = std::thread(spin_bool_foo, std::ref(flag));

    holder.join();
    worker.join();
}
```
std::atomic Spinlock Example

Holder releasing lock
Worker Spin lock opened doing stuff
Worker releasing lock
std::atomic<std::shared_ptr>
Atomic Builtins
Atomic Builtins

- Most Compilers have implementations for C++11 compatible atomic operations
- GCC and Clang use the `__atomic` prefix
Volatile vs Atomics
Volatile vs Atomics

```c
3  uint64_t sink;
4  
5  uint64_t func_vol() {
6    constexpr volatile uint64_t *ptr = &sink;
7    return *ptr;
8  }
9  
10 uint64_t func_atom() {
11    return __atomic_load_n(&sink, __ATOMIC_RELAXED);
12  }
```
X86-64 disassembly

```assembly
func_vol(): # @func_vol()
    mov    rax, qword ptr [rip + sink]
    ret

func_atom(): # @func_atom()
    mov    rax, qword ptr [rip + sink]
    ret

sink:
    .quad  0 # 0x0
```
X86-64 32 bits disassembly

```
1  func_vol():          # @func_vol()
2       call .L0$pb
3  .L0$pb:
4       pop  eax
5  .Ltmp0:
6       add  eax, offset _GLOBAL_OFFSET_TABLE_+(_.Ltmp0-.L0$pb)
7       mov  edx, dword ptr [eax + sink@GOTOFF+4]
8       mov  eax, dword ptr [eax + sink@GOTOFF]
9       ret
```
X86-64 32 bits disassembly

```
10 func_atom():               # @func_atom()
11    call .L1$p
12 .L1$p:
13    pop   eax
14 .Ltmp3:
15    add   eax, offset _GLOBAL_OFFSET_TABLE_+(.Ltmp3-.L1$p)
16    movq  xmm0, qword ptr [eax + sink@GOTOFF]  # xmm0 = mem[0],zero
17    movd  eax, xmm0
18    pshufd xmm0, xmm0, 85  # xmm0 = xmm0[1,1,1,1]
19    movd  edx, xmm0
20    ret
21   sink:
22    .quad 0            # 0x0
```
RISC V-64 disassembly

```assembly
func_vol():               # @func_vol()
  .Lpcrel_hi0:
    auipc    a0, %pcrel_hi(sink)
    addi     a0, a0, %pcrel_lo(.Lpcrel_hi0)
    ld       a0, 0(a0)
    ret

func_atom():            # @func_atom()
  .Lpcrel_hi1:
    auipc    a0, %pcrel_hi(sink)
    ld       a0, %pcrel_lo(.Lpcrel_hi1)(a0)
    ret

sink:                  # 0x0
  .quad  0
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky
RISC V-32 disassembly

```assembly
1. func_vol():
   lui a0, %hi(sink)
   lw a1, %lo(sink+4)(a0)
   lw a0, %lo(sink)(a0)
   ret

2. func_atom():
   addi sp, sp, -16
   sw ra, 12(sp)        # 4-byte Folded Spill
   lui a0, %hi(sink)
   addi a0, a0, %lo(sink)
   li a1, 0
   call __atomic_load_8
   lw ra, 12(sp)        # 4-byte Folded Reload
   addi sp, sp, 16
   ret
```
Fence Example
Relaxed

```cpp
std::atomic<int> flag{0};
uint64_t shared_val{};

void IncsharedXtimeNoAquire(){
    for (int count{0}; count < 10000000; ) {
        int expected{0};
        if (flag.compare_exchange_strong(expected, 1,
            std::memory_order_relaxed)){
            //lock successful
            shared_val++;
            count++;
            flag.store(0, std::memory_order_relaxed);
        }
    }
}
```
Relaxed

```assembly
1  IncsharedXtimeNoAquire():          # @IncsharedXtimeNoAquire()
2      xor    ecx, ecx
3      mov    rdx, qword ptr [rip + shared_val]
4      mov    esi, 1
5      jmp   .LBB0_1
6  .LBB0_3:                              # in Loop: Header=B0_1 Depth=1
7      cmp    ecx, 10000000
8      jge   .LBB0_4
9  .LBB0_1:                              # =>This Inner Loop Header: Depth=1
10     xor    eax, eax
11     lock   cmpxchg dword ptr [rip + flag], esi
12     jne   .LBB0_3
13     inc    rdx
14     mov    qword ptr [rip + shared_val], rdx
15     inc    ecx
16     mov    dword ptr [rip + flag], 0
17     jmp   .LBB0_3
18  .LBB0_4:
19      ret
```
Relaxed

```c
1     _IncsharedXtimeNoAquire():  # @IncsharedXtimeNoAquire()
2       xor   ecx, ecx
3       mov   rdx, qword ptr [rip + shared_val]
4       mov   esi, 1
5       jmp   .LBB0_1
6     .LBB0_3:  # in Loop: Header=LBB0_1 Depth=1
7         cmp   ecx, 10000000
8         jge   .LBB0_4
9     .LBB0_1:  # =>This Inner Loop Header: Depth=1
10        xor   eax, eax
11        lock  cmpxchg dword ptr [rip + flag], esi
12        jne   .LBB0_3
13        inc   rdx
14        mov   qword ptr [rip + shared_val], rdx
15        inc   ecx
16        mov   dword ptr [rip + flag], 0
17        jmp   .LBB0_3
18     .LBB0_4:
19        ret
```
Relaxed

```assembly
1  void IncsharedXtimeNoAquire(): # @IncsharedXtimeNoAquire()
2       xor    ecx, ecx
3       mov    rdx, qword ptr [rip + shared_val]
4       mov    esi, 1
5       jmp    .LBB0_1
6  .LBB0_3:  # in Loop: Header=LBB0_1 Depth=1
7       cmp    ecx, 10000000
8       jge    .LBB0_4
9  .LBB0_1:  # ->This Inner Loop Header: Depth=1
10      xor    eax, eax
11      lock   cmpxchg dword ptr [rip + flag], esi
12      jne    .LBB0_3
13      inc    rdx
14      mov    qword ptr [rip + shared_val], rdx
15      inc    ecx
16      mov    dword ptr [rip + flag], 0
17      jmp    .LBB0_3
18  .LBB0_4:
19      ret
```
Relaxed

```
1  IncsharedXtimeNoAquire(): # @IncsharedXtimeNoAquire()
2     xor   ecx, ecx
3     mov   rdx, qword ptr [rip + shared_val]
4     mov   esi, 1
5     jmp   .LBB0_1
6  .LBB0_3: # in Loop: Header=BB0_1 Depth=1
7     cmp   ecx, 10000000
8     jge   .LBB0_4
9  .LBB0_1: # =>This Inner Loop Header: Depth=1
10    xor   eax, eax
11    lock   cmpxchg dword ptr [rip + flag], esi
12    jne   .LBB0_3
13    inc   rdx
14    mov   qword ptr [rip + shared_val], rdx
15    inc   ecx
16    mov   dword ptr [rip + flag], 0
17    jmp   .LBB0_3
18  .LBB0_4:
19    ret
```
Relaxed surprise

run0  shared_val = 10266917
run1  shared_val = 10258969
run2  shared_val = 10174515
run3  shared_val = 10001845
```c
void IncsharedXtimeAquire()
{
    for (int count[0]; count < 10000000; ) {
        int expected[];
        if (flag.compare_exchange_strong(expected, 1, std::memory_order_acquire)) {
            //lock successful
            shared_val++;
            count++;
            flag.store(0, std::memory_order_release);
        }
    }
}
```
Acquire

```plaintext
.IncsrextmeAquire():  # @IncsharedXtimeAquire()
   xor  ecx, ecx
   mov  edx, 1
   jmp  .LBB1_1
.LBB1_3:               # in Loop: Header=BB1_1 Depth=1
   cmp  ecx, 10000000
   jge  .LBB1_4
.LBB1_1:               # =>This Inner Loop Header: Depth=1
   xor  eax, eax
   lock  cmpxchg dword ptr [rip + flag], edx
   jne  .LBB1_3
   inc  qword ptr [rip + shared_val]
   inc  ecx
   mov  dword ptr [rip + flag], 0
   jmp  .LBB1_3
.LBB1_4:
   ret
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky
Acquire No Surprises

```plaintext
run0  shared_val = 20000000
run1  shared_val = 20000000
run2  shared_val = 20000000
run3  shared_val = 20000000
```
RISC V Disassembly

IncsharedXtimeNoAquire():
li a4,9998336
addi a4,a4,1664
lui a3,%hi(flag)
lui a2,%hi(shared_val)
li a1,1
.L6:
addi a0,a3,%lo(flag)
1: 1r.w a5,0(a0); bne a5,zero,1f; sc.w a6,a1,0(a0); bnez a6,1b; 1:
sext.w a5,a5
bnez a5,.L6
ld a5,%lo(shared_val)(a2)
addi a5,a5,1
sd a5,%lo(shared_val)(a2)
addi a5,a3,%lo(flag)
amoswap.w zero,zero,0(a5)
addiw a4,a4,-1
bnez a4,.L6
ret
RISC V Disassembly

148  IncsharedXtimeAquire():
149      li   a4,9998336
150      addi a4,a4,1664
151      lui  a3,%hi(flag)
152      lui  a2,%hi(shared_val)
153      li   a1,1
154 .L27:
155      addi a0,a3,%lo(flag)
156      1: lr.w.aq a5,0(a0); bne a5,zero,1f; sc.w.aq a6,a1,0(a0); bnez a6,1b; 1:
157      sext.w a5,a5
158      bnez a5,.L27
159      ld  a5,%lo(shared_val)(a2)
160      addi a5,a5,1
161      sd  a5,%lo(shared_val)(a2)
162      addi a5,a3,%lo(flag)
163      fence iorw,ow; amoswap.w zero,zero,0(a5)
164      addiw a4,a4,-1
165      bnez a4,.L27
166      ret
Other STL Concurrency Features
std::future and std::promise

- std::future<T> - value for which you may wait
- std::promise<T> - produces a future

```cpp
int main()
{
    std::promise<int> p;
    std::thread t1([]&p){
        std::this_thread::sleep_for(1s);
        p.set_value(42);
    });

    std::thread t2([](auto f){
        fmt::print("The value is {}", f.get());
    }, p.get_future());

    t1.join();
t2.join();
}
```
Futures and Promises

- Can help communication between threads
- Help build task-oriented utilities for executing work on different threads
- Future and promise are one shoot operation
Jthread

- Same as std::thread but is joinable by default
- Jthread is stoppable with std::stop_source
- Provides easier implementation where user don’t have to think about joins
Synchronization Tools

- `std::mutex, std::conditional_variable`
- `std::lock_guard` – RAII helper for locking

```cpp
int main()
{
    std::mutex m;
    auto doer = [&m](int idx) {
        std::lock_guard l(m);
        fmt::print("doer{} thread doing stuff\n", idx);
    };
    std::jthread t1(doer, 0);
    std::jthread t2(doer, 1);
}
```
Synchronization Tools

- `std::unique_lock` – same as lock guard but unique and can be deferred.
- `std::scoped_lock` – takes ownership of multiple locks at once with RAII with deadlock avoidance algorithm
Synchronization Tools

- `std::counting_semaphore` – semaphore that can be set to an arbitrary number.
- `Std::binary_semaphore` – same as `counting_semaphore` but it's set to 1.
Synchronization Tools

```cpp
std::binary_semaphore start_speaking(0), person1_spoke(0), person2_spoke(0);

void person_speak(std::string_view text, std::binary_semaphore& w, std::binary_semaphore& r){
    w.acquire();
    fmt::print("{}"); text
    std::this_thread::sleep_for(std::chrono::seconds(1));
    r.release();
}

int main(){
    std::jthread p1(per
    std::jthread p2(per
    start_speaking.release();
    person2_spoke.acquire();
    fmt::print("GoodBye!");
    return 0;
}
Hello Person2 im Person1
Hello Person2 im Person1
Hello Person2 im Person1
Hello Person2 im Person1
Hello Person1 im Person2
Hello Person1 im Person2
Hello Person1 im Person2
Hello Person1 im Person2

Goodbye!
Synchronization Tools: stop_source

```c++
10 std::binary_semaphore start_speaking(0, p1_done[0], p2_done[0]);
11
12 void person_speak(std::stop_token s, std::string_view text, std::binary_semaphore & w, std::binary_semaphore & r){
13     w.acquire();
14     while (not s.stop_requested()){
15         fmt::print("{}"), text;
16         std::this_thread::sleep_for(1s);
17     }
18     r.release();
19 }
20
21 void stop_speaking(std::array<std::stop_source, 2> ssr){
22         for (auto i = 0u; i < ssr.size(); i++){
23             std::this_thread::sleep_for(2s);
24             ssr[i].request_stop();
25         }
26 }
27
28 int main() {
29     std::jthread p1(person_speak, "Hello Person2 im Person1\n",
30             std::ref(start_speaking), std::ref(p1_done));
31     std::jthread p2(person_speak, "Hello Person1 im Person2\n",
32             std::ref(p1_done), std::ref(p2_done));
33     start_speaking.release();
34     std::jthread sched(stop_speaking, std::array{p1.get_stop_source(), p2.get_stop_source()},
35             p2_done.acquire());
36     fmt::print("Goodbye!");
37 }
```
Std::threads : Micro Benchmark
Synchronization Tools - cont

- `std::latch`— threads may block on latch until its value is zero, latch is one shoot.
- `std::barrier`— very similar to latch but can be used multiple times.
Synchronization Tools – Latch Example

```cpp
int main()
{
    std::vector<std::string_view> ppl{"Alex", "Dani", "Benny", "Guy"};
    std::vector<std::jthread> workers;
    std::latch jobs{ppl.size()}, go_home{1};
    auto job = [&jobs, &go_home](std::string_view name){
        fmt::print("\{\} is doing a job\n", name);
        jobs.count_down();
        go_home.wait();
        fmt::print("\{\} is going home\n", name);
    };

    for (const auto& p : ppl) {
        workers.push_back(std::jthread(job, p));
    }
    jobs.wait();
    fmt::print("Go home!\n");
    go_home.count_down();
}
```
Synchronization Tools – Latch Example

Alex is doing a job
Dani is doing a job
Guy is doing a job
Benny is doing a job
Go home!
Alex is going home
Dani is going home
Guy is going home
Benny is going home
Synchronization Tools – Barrier Example

```cpp
int main()
{
    std::vector<std::string_view> ppl{"Alex", "Dani", "Benny", "Guy"};
    std::vector<std::jthread> workers;
    auto on_complete = [](){
        static int x = 0;
        not x++ ? fmt::print("Go Home!\n") : fmt::print("Done!\n");
    };
    std::barrier sync{ppl.size(), on_complete};
    auto job = [&sync](std::string_view name){
        fmt::print("{} is doing a job\n", name);
        sync.arrive_and_wait();
        fmt::print("{} is going home\n", name);
        sync.arrive_and_wait();
    };
    for (const auto& p : ppl) {
        workers.push_back(std::jthread(job, p));
    }
}
```
Synchronization Tools – Barrier Example

Dani is doing a job
Guy is doing a job
Benny is doing a job
Alex is doing a job
Go Home!
Benny is going home
Alex is going home
Dani is going home
Guy is going home
Done!
Synchronization Tools

- `std::shared_mutex` – may be used by couple of threads or be exclusive.
- `std::timed_mutex` – same as mutex but has a claim time out.
- `std::shared_timed_mutex` – combination of shared and timed mutexes.
- `std::shared_lock` – RAII wrapper for timed and shared mutexes.
Synchronization Tools

```cpp
int val = 10;

int main()
{
    std::shared_timed_mutex stm;
    auto access_and_print = [&stm](int v){
        std::shared_lock l sts
        my_val = val + v;
        fmt::print("val {}
        
    });

    std::jthread t1(access_and_print, 1);
    std::jthread t2(access_and_print, 2);
}
```
Synchronization Tools

- Static: from C++11 static variables initialization is magic

```c
struct X{
    int x;
};

global_array get_once(int): # @get_once(int)
    movzx eax, byte ptr [rip + guard variable for get_once(int)::single]
    test al, al
    je .LBB3_1
    lea rax, [rip + get_once(int)::single]
    ret

    std::jthread t1([](){auto& x = get_once(10);}),
    t2([](){auto& x = get_once(11);});
```
Synchronization Tools

- `std::call_once` and `std::once_flag` – help to do something just once.

```cpp
int main()
{
    std::once_flag flag;
    auto f = [&flag](){
        static int x{10};
        std::call_once(flag, [](){x++;});
        fmt::print("{}\n", x);
    };

    std::jthread t1(f);
    std::jthread t2(f);
}
```
Synchronization Tools

- `std::packaged_task` – wraps a callable into a thread and returns a future

```cpp
#include <iostream>
#include <future>

int main(){
    std::packaged_task<int(int, int)> sum_task([](int a, int b){
        return a+b;
    });

    std::packaged_task<int(int, int)> mul_task([](int a, int b){
        return a*b;
    });

    mul_task(10, 2);
    sum_task(10, mul_task.get_future().get());

    std::cout << "tot: {}", sum_task.get_future().get();

    return 0;
}
```
QUESTIONS
THANK YOU FOR LISTENING

ALEX DATSKOVSKY
+97254-7685001
ALEX.DATSKOVSKY@SPEEDATA.IO
WWW.LINKEDIN.COM/IN/ALEXDATHSKOVSKY
https://www.cppnext.com