MEMORY MODEL

Introduction

Few Words About me

- **Alex Dathskovsky**

- Developer with experience of more than 14 years.
- Performance expert
- Linux environment expert
- Architect for large scale systems

- Contact: Mail - calebxyz@gmail.com
- Phone – 0547685001

- Question
  - Does the computer execute the program you wrote?

- Answer
  - No
    - Compilers , Processors reorder the operations
      - Compilers reorder software
      - CPU’S use OOO execution.

- Compiler Reordering
  - Consider the next code (what would you do?)

```
int sum = 0;
int p = get_pivot();
for (int i(0); i < p; i++)
{
	sum += data[i] * data[p];
}
```

  - The compiler sees that you read the same value over and over so he takes it out of the loop to optimize it.

- Compiler Reordering
  - The compiler automatically vectorize your code
  - Reordering RW Operations

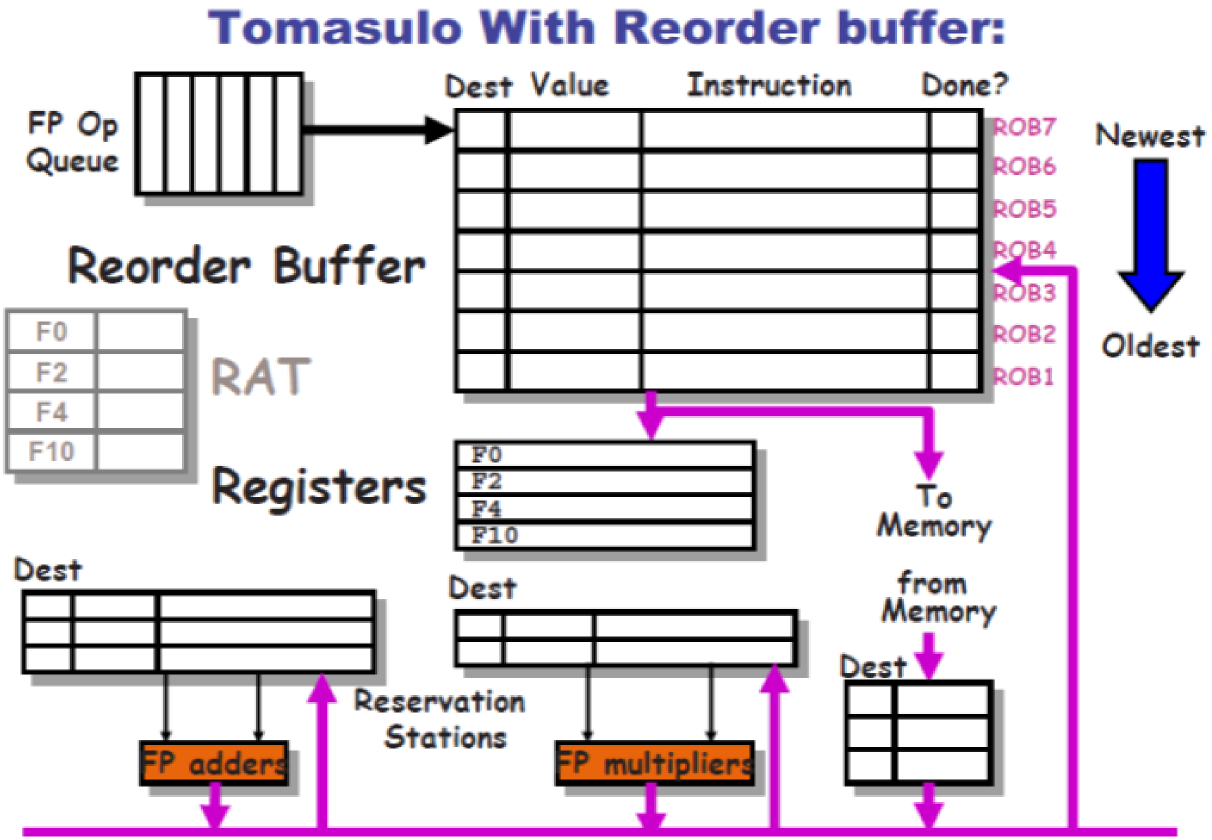
```
void enqueue(T newElem) {
    que[++last] = newElem;
    locked = 0; };

void enqueue(T newElem) {
    locked = 0;
    que[++last] = newElem; };
```

OOO Processors

Alex Dathskovsky

- Processor Reordering
  - The processor use OOO execution.



- Difference between in order and out of order
  - In Order:
    1) Instruction fetch
    2) If operands available execute if not fetch them.
    3) The instruction is executed by the functional unit
    4) The functional unit writes the result back to the register
  - Out of Order:
    1) Instruction fetch
    2) Instruction dispatch to an instruction queue
    3) The instruction waits in the queue until its input operands are available. The instruction is then allowed to leave the queue before earlier, older instructions.
    4) The instruction is issued to the appropriate functional unit and executed by that unit
    5) The results are queued
    6) Only after all older instructions have their results written back to the register file, then this result is written back to the register file. This is called the graduation or retire stage

- Dynamic Scheduling
  - check for structural hazards
    - an instruction can be issued when a functional unit is available
    - an instruction stalls if no appropriate functional unit
  - check for data hazards
    - an instruction can execute when its operands have been calculated or loaded from memory
    - an instruction stalls if operands are not available

- Dynamic Scheduling
  - don’t wait for previous instructions to execute if this instruction does not depend on them, i.e., independent ready instructions can execute before earlier instructions that are stalled check for data hazards
  - ready instructions can execute before earlier instructions that are stalled

- Dynamic Scheduling
  - In order:

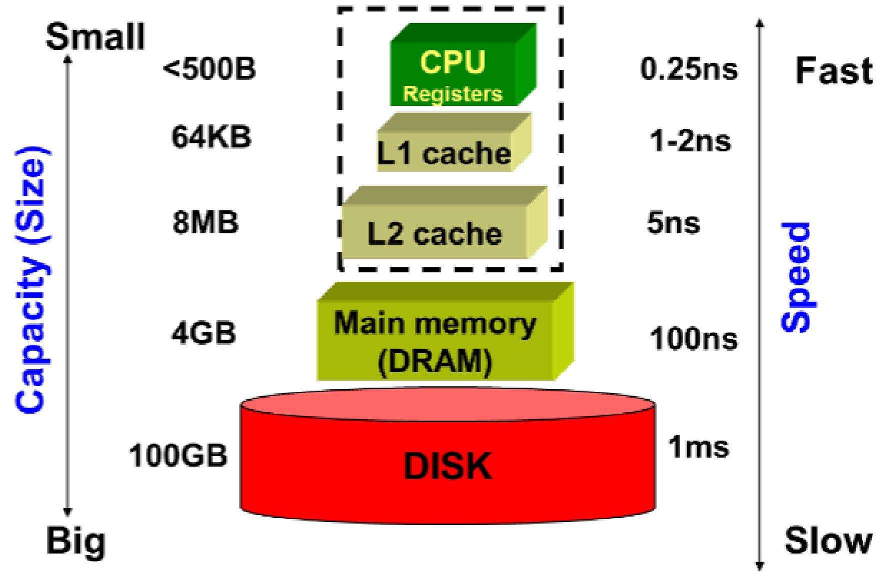
lw $3, 100($4)	in execution, cache miss
add $2, $3, $4	waits until the miss is satisfied
sub $5, $6, $7	waits for the add

  - Out of order

lw $3, 100($4)	in execution, cache miss sub
sub $5, $6, $7	can execute during the cache miss add
add $2, $3, $4	waits until the miss is satisfied

- Speculative Execution
  - Instruction speculation: executing an instruction before it is known that it should be executed
    - all instructions that are fetched because of a prediction are speculative

- Cache
  - Cache and write buffers can dramatically delay memory writes and speedup memory reads



CPU registers are the highest level of memory, part of the ISA

Reordering in C++

- Reordering types
  - Data dependencies must be honored
  - C++ compiler may reorder any memory access under the as-if rule
  - Different processes have different reordering guaranties

- AS-IF Rule

  1) 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.
  2) At program termination, data written to files is exactly as if the program was executed as written.
  3) Prompting text which is sent to interactive devices will be shown before the program waits for input

- AS-IF But not always
    1) Invalid access to an array , writing to a constant and more .
    2) Loading an external library.
    3) copy elision

- # Reordering Example

- Thread 1

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

Thread2

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

- Store can pass loads so both threads see other flag as zero and enter the critical section

- Sequential Consistency
  - the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program
  - SC is very expensive
  - Modern compilers do not offer it.

- SC-DRF
  - Race Conditions
    - A memory location can be accessed simultaneously by two threads, one of which is a writer
  - SC for data race free programs
    - Executing reads and writes in program order, as long as there is no race condition
    - Hardware promises SC if you obey the constrains and do not write race conditions.
  - **C++11 Offers SC-DRF**

- Volatile
  - Volatile prevents the compiler from reordering the reads and writes.
  - Volatile prevents register caching.
  - Volatile don’t prevent hardware OOO.

- Compiler code barriers
  - Prevent compiler from moving reads or writes across the barrier.

Thread 1

```
Value = very_long_calc()
asm  volatile("" ::: "memory")
done = true
```

Thread2

```
if (done)
asm  volatile("" ::: "memory")
do something
```

- Processor memory barriers
  - Prevent all reads and writes from passing the barrier.

Thread 1

```
If (ptr != nullptr)
Ptr -> do_work()
```

Thread2

```
auto tmp = new exp();
mb();
ptr = tmp;
```

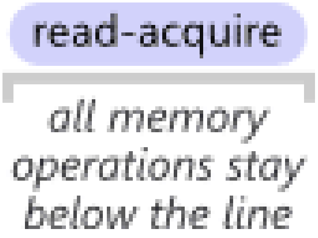
- Locks and Mutexes
  - Operation on sync mechanisms are memory barriers
  - Operation on std::atomic variables are memory barriers

- For more info and examples

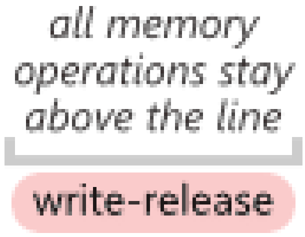
- http://preshing.com/20120515/memory-reordering-caught-in-the-act/

Acquire and Release Semantics

- 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.*



- 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.*



Concurrency Lib

- atomic
  - Provides a portable way to perform low-level atomic memory operations
    - Atomic: no torn reads or torn writes
    - Ordered: acquire/release and additional memory ordering guarantees

- Atomic Memory Orders
  - memory_order_relaxed -
    - Relaxed operation: there are no synchronization or ordering constraints imposed on other reads or writes, only this operation's atomicity is guaranteed
  - memory_order_acquire – we know this one
  - memory_order_release – we know this one
  - memory_order_seq_cst – default
    - All load operations are read acquire all store operations are write release, read modify write perform both

- # Atomic with memory fences

```
int A = 0;
atomic<int> Ready{0};
```

Thread 1

```
A = 42
atomic_thread_fence(memory_order_release);
Ready.store(1, memory_order_relaxed);
```

Thread 2

```
int r1 = Ready.load(memory_order_relaxed);
atomic_thread_fence(memory_order_acquire)
int r2 = A;
```

- # Atomic Without Fences

```
int A = 0;
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;
```

- Now to our old example

```
std::atomic<int> flag1{0} , flag2{0}
```

- Thread 1

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

Thread2

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

- Performance
  - Loads should be fast
    - On x86 x64 atomic loads are regular loads
    - Can be more expensive on other platforms
  - Stores can be slower
    - On x86/64 atomic stores use xchg , full barrier

- Example

```
class spinlock
{ std::atomic<bool> m_lock{false};
  public:
  void lock(){
     while (m_lock.exchange(true)) }
  void unlock(){
           m_lock = false; } }
```

- thread
  - This class represents a cross platform thread
    - Can be created.
    - Wait for completion
    - Get Native Hndl.
  - Example

```
thread t([](){this_thread::sleep_for(5s); cout<< done;)});
t.join();
```

- Futures and promises
  - Std::future<T> represent a future value for which you can wait patiently
  - Std::promise<T> produces a future
  - Example:

```
std::promise<bool> prom; auto future = prom.get_future();
thread t([&future](){cout << future.get();});
prom.set_value(42);
```

- Futures and promises
  - Can help marshal values and exceptions across threads
  - Help build task oriented utilities for executing work in different threads
  - Example:

```
std::promise<bool> prom; auto future = prom.get_future();
std::vector<int> numbers {…….};
auto sumTrd = std::thread([&prom, &numbers](){
auto sum = std::accumulate(numbers.begin(), numbers.end(), 0), prom.set_value(sum)});
auto doTrd = std::thread([&future](){ //do something
auto sum = future.get(); // do somthing});
```

- async
  - A helper method to create a future based async task
    - May create a thread for each invocation
  - Example:

```
Template <typename Titer>
int parallelSum(Titer beg, Titer end)
{ auto len = end – beg;
   if (len < 1000) return std::accumulate(beg,end,0);
   Titer mid = beg + len /2
    auto hndl = std::async(std::lanch::async, parallelSum<Titer>, mid, end)
    int sum = parallelSum(beg,mid);
    return sum+ handle.get();}

Std::vector<int> v(10000, 1);
STD::COUT << “THE SUM IS” << parallerSum(v.begin(), v.end());
```

- Async pitfall
  - std::future waits for the value to become available.
    - May take time
  - Std::async returns a future if you don’t save it or outlive it it’s a temp that expires
  - Bad Practice:

```
async(do_this);
async(do_that);
async(do_more);
```

Will run sequentially, no parallelism!

- Synchronization
  - Std::mutex, std::conditional_variable
  - Std::lock_guard is a RAII helper for locking

- Conditional Variables and Mutex by example

```
std::queue<std::string> que;
std::mutex lock;
std::conditional_variable workReady;

void produce(std::string url) {
	std::unique_lock<std::mutex> ul(lock);
	que.push(url);
	if (que.seze() == 1) workReady.notify_one(); }

void consume() {
	std::unique_lock<std::mutex> ul(lock);
	workRead.wait(lk, [](){return que.size() > 0});
	browse(que.front()); que.pop();
}
```

- Singleton
  - In C++11 the static became magical

```
T& getReference()
{
    static T ref;
    return ref;
}
```

- timed_mutex
- Same as the good old mutex but with a timeout option
- try_lock_for – try to acquire the lock for time
- try_lock_until – try to lock until time is reached

- recursive_mutex, recursive_timed_mutex
- Same thread may call the lock couple of times and the lock will be released after same number of unlocks was called

- shared_timed_mutex
- Shared mutexes are usually used in situations when multiple readers can access the same resource at the same time without causing data races, but only one writer can do so.
- Shared _mutex (appeared in c++17)

- shared_timed_mutex example

```
std::shared_timed_mutex m;
my_data_structure data;

void reader(){
    std::shared_lock<std::shared_timed_mutex> lk(m);
    do_something_with(data);
}

void writer(){
    std::lock_guard<std::shared_timed_mutex> lk(m);
    update(data);
}
```

- Lock, try_lock
- Locks as many lockable objects as you need

```
template< class Lockable1, class Lockable2, class... LockableN >
void lock( Lockable1& lock1, Lockable2& lock2, LockableN&... lockn );
```

- call_once
- call an operation once from all threads

```
#include <mutex>

std::once_flag flag;

void doSomthing()
{
    ;;;
}

void threadMain()
{
    std::call_once(flag, doSomthing);

    ;;;
}
```

- packaged_task
- Task that wraps any callable object and returns a future

```
#include <math.h>
#include <future>
#include <iostream>

void task_lambda()
{
    std::packaged_task<int(int,int)> task([](int a, int b)
    { return std::pow(a, b); });
    std::future<int> result = task.get_future();
    task(2, 9);
    std::cout << "task_lambda:\t" << result.get() << '\n';
}
```

- More please
  - shared_future

- Bonus Example
  - This code doesn't use the acquire release semantics

```
#include <atomic>
#include <thread>
#include <random>
#include <chrono>

std::atomic<int> flag;
int sharedValue;
std::mt19937_64 eng{std::random_device{}()};
std::uniform_int_distribution<> dist{100, 500};
```

- Bonus Example
  - This code doesn't use the acquire release semantics

```
void IncrementSharedValue10000000TimesNoAqcuire()
{
    int count = 0;
    while (count < 10000000)
    {
        std::this_thread::sleep_for(std::chrono::milliseconds{dist(eng)});
        int expected = 0;
        if (flag.compare_exchange_strong(expected, 1,
                std::memory_order_relaxed))
        {
            // Lock was successful
            sharedValue++;
            flag.store(0, std::memory_order_relaxed);
            count++;
        }
    }
}
```

- Bonus Example
  - Disassembly

```
0x519fe:  mov    r0, r10
0x51a00:  bl     0x51dd4
0x51a04:  ldrex  r0, [r6]
0x51a08:  cmp    r0, r5
0x51a0a:  bne    0x51a14
0x51a0c:  strex  r1, r4, [r6]
0x51a10:  cmp    r1, #0
0x51a12:  bne    0x51a04
0x51a14:  cmp    r0, #0
0x51a16:  bne    0x519fe
0x51a18:  ldr.w  r0, [r11]
0x51a1c:  add.w  r8, r8, #1
0x51a20:  adds   r0, #1
0x51a22:  str.w  r0, [r11]
0x51a26:  str    r5, [r6]
0x51a28:  movw   r0, #38527
0x51a2c:  movt   r0, #152
0x51a30:  cmp    r8, r0
0x51a32:  ble    0x519fe
```

flag.compare_exchange_strong(...)

sharedValue++

flag.store(...)

    - Lucky us the compiler didn't reorder

- Bonus Example
  - Surprise!

```
Target Output
is_lock_free: true
sharedValue=19986385
sharedValue=19987222
sharedValue=19987136
sharedValue=19986333
sharedValue=19980460
```

- Possible reorder
- the memory interaction of str.w r0, [r11](the store to sharedValue) could be reordered with that of str r5, [r6] (the store of 0 to flag). In other words, the mutex could be effectively unlocked before we're finished with it!

- Bonus Example
  - This code use the acquire release semantics

```
void IncrementSharedValue10000000TimesAqcuire()
{
    int count = 0;
    while (count < 10000000)
    {
        std::this_thread::sleep_for(std::chrono::milliseconds{dist(eng)});
        int expected = 0;
        if (flag.compare_exchange_strong(expected, 1,
                std::memory_order_acquire))
        {
            // Lock was successful
            sharedValue++;
            flag.store(0, std::memory_order_release);
            count++;
        }
    }
}
```

- Bonus Example
  - Disassembly

```
0x4a9f6:  mov     r0, r10
0x4a9f8:  bl      0x4add4
0x4a9fc:  ldrex   r0, [r6]
0x4aa00:  cmp     r0, r5
0x4aa02:  bne     0x4aa0c              flag.compare_exchange_strong(...)
0x4aa04:  strex   r1, r4, [r6]
0x4aa08:  cmp     r1, #0
0x4aa0a:  bne     0x4a9fc
0x4aa0c:  dmb     ish
0x4aa10:  cmp     r0, #0
0x4aa12:  bne     0x4a9f6
0x4aa14:  ldr.w   r0, [r11]
0x4aa18:  add.w   r8, r8, #1           sharedValue++
0x4aa1c:  adds    r0, #1
0x4aa1e:  str.w   r0, [r11]
0x4aa22:  dmb     ish
0x4aa26:  str     r5, [r6]             flag.store(...)
0x4aa28:  movw    r0, #38527
0x4aa2c:  movt    r0, #152
0x4aa30:  cmp     r8, r0
0x4aa32:  ble     0x4a9f6
```

- Bonus Example
  - This time it works

Target Output

```
is_lock_free: true
sharedValue=20000000
sharedValue=20000000
sharedValue=20000000
sharedValue=20000000
sharedValue=20000000
```

Thank You