Low Latency Programming

How to measure time?

- build with release mode `>> cmake -DCMAKE_BUILD_TYPE=Release ../..`
- set cpu frequency to highest `>> sudo cpufreq-set -d 4.2Ghz`
- timer resolution is about 15ns `timespec ts; clock_gettime(CLOCK_MONOTONIC, ts);`
- consistent in my Yubo ubuntu machine (for both debug or release mode)

Part A. Programming tips

A1. Object copy

- prefer pass by reference (pointer, referece or `std::reference_wrapper`) to pass by value
  prefer move semantic to copy semantic, prefer emplace to move semantics, prefer reference to emplace
- prefer stack memory to heap memory (refer to `YLib` containers)
- avoid smart pointer
- avoid node based container
- reserve vector to avoid resize, reserve unordered set/map to avoid rehash

A2. Container and algorithm

- pick correct container (O(1) in hashmap vs O(logN) in map)
- sort data in container before repeated-search
- pick correct sorting (such as pigeonhole sort for integers)
- prefer bisection to linear search in general
  prefer linear search to bisection for search in smaller contiguous vector
- prefer iteration to recursion when subproblems overlap (such as fibonacci)
  iteration or recursion are fine when subproblems do not overlap (such as factorial)

A3. C++ skills

- for switch statement, sort cases by decreasing occurance probability
- prefer prefix increment to suffix increment
- prefer function overload to type conversion
- avoid runtime polymorphism : (1) two redirections, (2) non POD thus no `memcpy`, (3) branch prediction
- compile time calculation : (1) inline, (2) template, (3) `constexr/consteval/constinit` (may lead to bulky code and thrashing)

A4. Calculation

- use integers for calculaton (grey level, price level)
- use sparse matrix instead of ordinary matrix if possible
- replace pow by multiplication
- replace multiplication of 2 by bit shift
- calculation using MMX / SSE /AVX
- mathematical function using lookup table
- mathematical function using approximation with Taylor series
- cache and incremental update of mean and variance

A5. On the hot path (time critical path)

- no allocation, no deallocation, no vector resize, no copying
- no mutex, no condition variable
- no file *IO*, no socket *IO*, no pipe *IO*

Part B. Cache friendly coding (reduce cache miss and ping pong)

`B1.` Avoid ***data cache-miss*** (single thread)

- putting correlated members together in class definition (order of members does make huge difference)
- vector scanning : linear search may be faster than binary search for small vector
- matrix scanning : row-major scan vs column-major scan
- matrix loop-blocking technique
- use container which stores elements in contiguous memory

-(1) prefer vector to list — if no insertion in the middle is needed

-(2) prefer ***hashmap*** to ***binary search tree*** — if no sorting is needed (but still offer searching)

`B2.` Avoid ***instruction cache-miss*** (single thread)

- avoid unpredictable branch by converting ***if-statement*** into ***non-if-statement***
- avoid unpredictable branch by avoiding vtable caused by virtual function
- avoid unpredictable branch by sorting data if repeated iterations are needed
- using `gcc` built-in function `__builtin_expect`

`B3.` Avoid cache line sharing

- line sharing leads to frequent cache-miss — solved by : ***set-associative-cache*** instead of ***direct-mapped-cache***, done in *CPU*
- true sharing leads to ***cache ping-pong*** — solved by : code review so that no two threads access the same address
- false sharing leads to ***cache ping-pong*** — solved by : (1) ***padding*** in `next_write`, (2) ***padding*** in `tick`, (3) division of labour

`B4.` Avoid paging and thrashing (single thread)

- avoid lots of `inline` resulting in bulk program and frequent ***page-fault*** when accesing *virtual memory space*
- avoid paging using `mlock`

memory lock vs ***memory barrier***

Part C. Concurrency – thread vs process

`C1.` Multi-thread

- various locks — (1) mutex vs shared mutex, (2) mutex vs spinlock, (3) mutex vs semaphore
- various models — (1) producer consumer model, (2) mapreduce with threadpool, (3) disruptor pattern
- blocking vs non blocking — (1) blocking, (2) lockfree, (3) waitfree

`C2.` Avoid context switch

- setting ***affinity*** — pinning a thread to a *CPU* *(affinity is for thread)*
- setting ***nice value / priority / sche policy*** — dedicating a *CPU* to a process *(priority is for process)*
- setting ***isolation*** in *BIOS* — isolating the *CPU* from *OS* scheduler

Part D. Input/Output and TCP socket

- write to file or database with separate thread
- design efficient protocol (optimal throughput vs optimal parsing time)
- TCP socket : disable Nagle algorithm
- TCP socket : async vs sync (which is faster?)
- TCP offload and kernel bypass (2 different features offered by 10G network card like Solarflare) :

- by installing new network card and driver, run both the network card driver and the application in user space, then ...
- all message exchanges between network card and application are done directly without going though kernel, besides ...
- all TCP checksum that is originally done by kernel are now delegated to the network card

Part F. Generic parser in TraderRun 444

F1. TraderRun parser is generic :

- support parsing and composing
- support various protocol
- support various input stream (file, tcp, udp etc)
- support various action (insert into lockfree buffer, log into file, write into database)

F2. TraderRun parser is fast :

- each tick is aggregate, memcpy can be used
- each tick is copied twice : once from socket to buffer, once from buffer directly to tick inside lockfree container
- skip undesired message, skip unwanted securities (no need to build orderbooks for unwanted securities)
- fire action on the run once a tick is done, no need to wait for parsing whole packet (1 packet = snapshots multiple stocks)

F3. Relation among :

- parser
- protocol definition
- stream (contains socket & buffer)
- action (contains lockfree buffer & tick)

main flow

```
cdoi::parser<protocol> parser(io, tcp_stream, action);
parser.async_run();
std::thread t(&io_service::run, std:ref(io_service));
```

inside parser async run

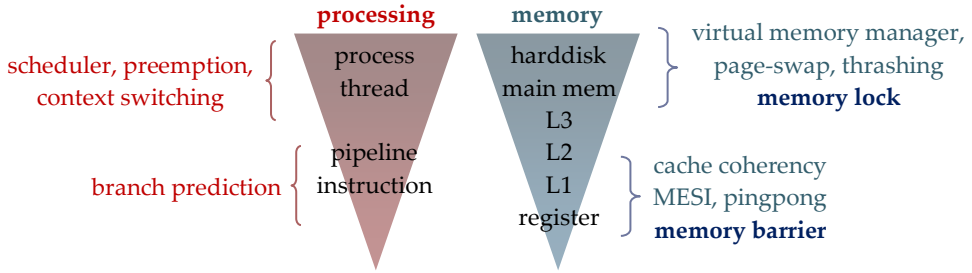
```
auto buffer = tcp_stream.get_buffer();
tcp_stream.socket.async_read(buffer, callback = action);
```

inside action

```
while (tcp_stream.contain_unparsed_data())
{
     auto& tick = action.lockfree_buffer.get_next_tick();
     if (tick.deserialize_from(tcp_stream))
          action.lockfree_buffer.next_tick_done();
     else break; // incomplete tick in last socket read
}
parser.async_run();
```

Part K. Mechanic sympathy

K1.	computer system	3 socket / core / hyperthread
K2.	file descriptor	5 in-out-err / file / memory / socket / pipe
K3.	process and thread	3 process / thread / differences
K4.	semaphore and mutex	3 semaphore / problems / mutex
A2.	MMX / SSE / AVX	
Bi	what is cache? and cache hierarchy	2
Bii	cache hit-miss and locality of reference	2
Biii	cache line and cache associativity	3 cache line / direct-map / set-associativity
Biv	cache coherency	3 MESI / store buf and invalid queue / memory order
B1.	avoid data cache-miss	2 naive matrix multiplication / loop blocking technique
B2.	avoid instruction cache-miss	5 branch prediction / sort data / remove if / remove vtable / builtin expect
B3.	avoid cache line sharing	4 line sharing / cache ping pong / true sharing / false sharing
B4.	avoid thrashing	3 VAS and paging / thrashing / avoid paging and thrashing `mlock()`
C2.	avoid context switch	4 context switch / solution 1 / solution 2 / solution 3
D.	TCP socket	5 TCP vs UDP
		2 TCP using UDP
		1 Nagle



Mechanical Sympathy

My *cdoi* machine is single Intel® Xeon® 3.47GHz cpu with 6 cores and 4 hyperthreads, thus 24 logical *CPUs*, with 12,288KB cache.

K1. Computer system

Here is a typical system configuration :

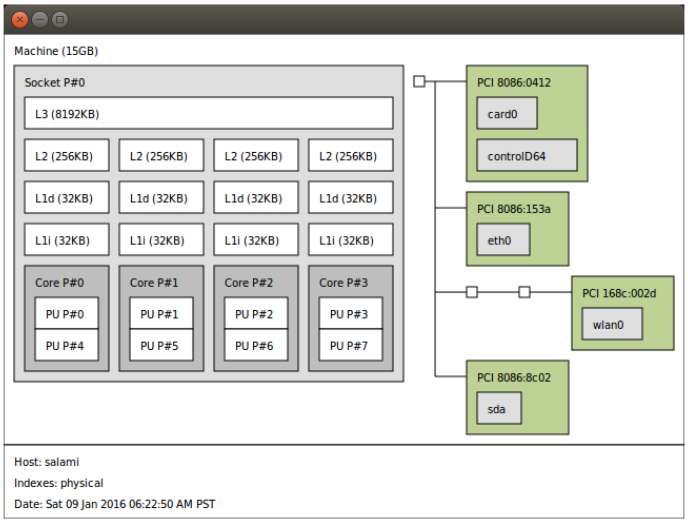
- each machine has multiple ***CPU sockets***, on which *CPU* chips can be inserted
- each CPU has multiple ***cores***, each has its own execution resources, *L1 L2* cache, but share the same *L3* cache
- each core interleaves multiple ***hyperthreads*** (also called logical *CPU)* they share execution resources and all caches

Please read *"C++11 threads, affinity and hyperthreading" by Eli Bendersky*. In linux, use command `lscpu` to list system information with text or command `lstopo` to display system information with graphics :

```
$ lscpu

Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 60
Stepping:              3
CPU MHz:               3501.000
BogoMIPS:              6984.09
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
```

```
$ lstopo
```



K2. File descriptor

File descriptor `FD` are handles to *IO* objects that support `open/close/read/write`, including :

- standard input / output /error
- file
- shared memory
- network socket
- named pipe and anonymous pipe

We can map the physical memory of disk file or other IO devices (pointed by file descriptor) to virtual address space of the current process calling `mmap`. Thus we can access IO quickly by reading or writing to that *VAS* memory location.

```
*void mmap(void* addr, size_t size, int prot, int flags, int fd, off_t offset);

addr   = starting memory in VAS for mapping (if it is NULL, the system will find a location in VAS and returned by function mmap)
size   = size of  memory in VAS for mapping
prot   = PROT_READ / PROT_WRITE / PROT_EXEC
flag   = MAP_SHARED
fd     = file descriptor return from ::fopen or ::shm_open ... etc
offset = offser from the beginning of file descriptor, usually set zero
```

List of functions :

- for forking and getting process id — `fork, getpid`
- for opening shared memory as file descriptor — `shm_open, shm_unlink`
- for mapping file descriptor to memory — `mmap, unmmap`
- for truncating file descriptor size — `ftruncate`

Lets try shared memory here. Please link shared library `librt.so` (real time extension library) for definition of `shm_open`.

```
// link with option -lrt
#include <iostream>
#include <cstring>    // for memcpy and memset
#include <string>
#include <unistd.h>   // for getpid and fork
#include <fcntl.h>    // for shared mem
#include <sys/mman.h> // for mmap

void test_ipc()
{
    if (fork() > 0)
    {
        ipc::producer();
    }
    else
    {
        ipc::consumer();
    }
}

namespace ipc
{
    const std::string buffer_name = "ABC";
    const int buffer_size = 1024;

    void producer()
    {
        pid_t pid = getpid();
        std::cout << "\npid" << pid << " is parent running producer. " << std::flush;

        auto fd   = ::shm_open(buffer_name.c_str(), O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
        auto res  = ::ftruncate(fd, buffer_size); // truncate buffer size to fixed length
        auto addr = ::mmap(NULL, buffer_size, PROT_WRITE, MAP_SHARED, fd, 0);
        std::memset(addr, 0, buffer_size);
        std::uint8_t count = 0;

        while(true)
        {
            std::cout << "\nEnter command : ";
            std::string str;
            std::cin >> str;
            std::uint8_t length = (std::uint8_t)str.size();

            ++count;
            std::memcpy(addr+2, str.c_str(), str.size());
            std::memcpy(addr+1, &length, 1);
            std::memcpy(addr+0, &count, 1);

            if (str == "quit") break;
        }
        res = ::munmap(addr, buffer_size);
        fd  = ::shm_unlink(buffer_name.c_str());
    }

    void consumer()
    {
        pid_t pid = getpid();
        std::cout << "\npid" << pid << " is child running consumer. " << std::flush;

        auto fd   = ::shm_open(buffer_name.c_str(), O_RDONLY, S_IRUSR | S_IWUSR);
        auto addr = ::mmap(NULL, buffer_size, PROT_READ, MAP_SHARED, fd, 0);
        std::uint8_t count = 0;

        while(true)
        {
            if (reinterpret_cast<std::uint8_t*>(addr)[0] == count+1)
            {
                ++count;
                std::uint8_t length = reinterpret_cast<std::uint8_t*>(addr)[1];
                std::string str(reinterpret_cast<char*>(addr)+2, length);
                std::cout << "\nreceive msg" << (int)count << " : " << str << "\n" << std::flush;
                if (str == "quit") break;
            }
        }
    }
}
```

Question : How can we ensure that the shared memory is already created by producer before consumer reads it?

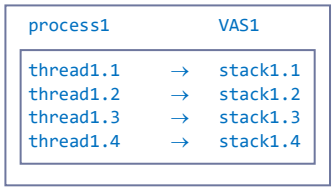
K3. Process vs thread

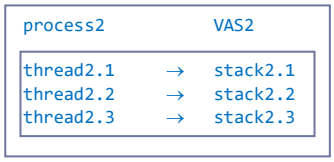
Process is an executing instance of a program, it has all resources for execution. Each process has individual :

- process id
- executable code
- *virtual address space*
- *environment variables*
- *file descriptor table*
- *signal handling table*

Thread is an small entity within a process :

- each thread has individual thread id, individual call stack
- all threads of the same process share *virtual address space, environment variables, file descriptor table* and *signal handling table*





Differences between process and thread :

- All threads in same process share the same heap memory, each thread has its own stack memory.
- Processes communicate via 6 interprocess communication *IPCs*, like *file, DB, socket, pipe, zero message queue, shared memory*. Threads communicate via shared variables in the same heap memory.
- Thread cannot be killed individually by user. Process does.

About process

Types of processes :

1. foreground = interactive process, created by user through terminal, ***NOT*** created automatically as part of system
2. background = non-interactive process, created by fork ... etc, ***NO*** user input
3. daemons = background process created during system start up and runs forever as a service

State of process and possible state transition :

1. creation ► ready
2. termination ► null
3. ready ► running (this transition happens when scheduler dispatches)
4. running ► waiting (this transition happens when *CPU* waits for *IO* or waits for event)
   ► ready (this transition happens when interrupt)
   ► terminated (this transition happens when program exits)
5. waiting ► ready (this transition happens when *IO* is done or event is done)

More about process :

- process creation means copy itself (instruction and data) from harddisk to memory
- process creation in C++ can be done using one of the functions : `system()`, `exec()` or `fork()`
- process is identified by `pid` and `ppid` (parent process id) which can be checked by command `pidof my_executable_name`
- `init` process is the mother of all processes in linux, 1st process created since system boot
- there is a queue of ready processes and a queue of waiting processes, requesting clock cycle (CPU resource) from scheduler

Preemptive and cooperative scheduling

- scheduling is allocation of cpu-clock-cycle (resource allocation) to processes

1. preemptive = OS as a cpu resource manager, can interrupt running process to run another, resulting in context switch
2. cooperative = OS does not preempt process, current process must cooperate with other processes by yielding willingly

Various kinds of algo, depending on : `https://afteracademy.com/blog/process-scheduling-algorithms-in-the-operating-system`

- created time (FIFO)
- burst time (shortest running time)
- nice value (highest priority first)

K4. Semaphore and Mutex *This section is a summary of "Mutex vs Semaphore", by Sticky Bits.*

What is semaphore?

Semaphore is introduced by Dijkstra in 1965, far before invention of mutex in 1980.

```
semaphore s(3); // N=3
void functor()
{
     s.decrement();
     do_something_in_critical_session();
     s.increment();
}
std::thread t0(functor);
std::thread t1(functor);
```

- Semaphore is a counter initialized as *N*, with an atomic `decrement()` and an atomic `increment()` function.
- Critical session is wrapped with `decrement()` at the beginning and `increment()` at the end, it allows *N* threads to enter :

- *when a thread enters critical session, it invokes* `decrement()`
  if the counter is non-zero, decrease it by 1, the thread then gets inside critical session
  if the counter is zero, the thread then waits to be notified in a queue
- *when a thread exits critical session, it invokes* `increment()` *and notify a waiting thread*

- Semaphore does not offer ownership, which means :

- *one thread can call* `increment()` *without calling* `decrement()` *in prior, semaphore doesn't do any checking, it means …*
- *one thread can notify waiting thread even if the former has never entered the critical session*

Problems without ownership

- miss-use of semaphore as ***signal or synchronization*** *(this is legal, but not a good practice)*

```
semaphore s(0);

// thread 1                                  // thread 2
s.decrement();                               do_something2();
do_something1();                             s.increment(); // notify thread 1
```

- accidental release or notification, due to bug-fix, cut-and-paste mistake

```
semaphore s(10);

// thread 1                                  // thread 2
s.decrement();                           //  s.decrement(); // commented during bug-fix, but forget to resume
do_something  _in_critical_session();        do_something_in_critical_session();
s.increment();                               s.increment();
```

- recursive deadlock, as semaphore does not check whether the same thread has called `decrement()` before

```
semaphore s(10);

void function()
{
     s.decrement();
     do_something  _in_critical_session();
     function();
     s.increment();
}
```

Mutex comes to rescue. By offering ownership, mutex can only release lock that it owns, otherwise there will be exception. With the ownership checking, all three problems above : ***miss-use as signal***, ***accidental release*** and ***recursive deadlock*** can be avoided.

```
mutex m;
void functor()
{
     m.request_lock();
     do_something_in_critical_session();
     m.release_lock();
}
std::thread t0(functor);
std::thread t1(functor);
```

Differences between mutex and semaphore

- mutex is atomic boolean, semaphore is atomic counter
- mutex offers ownership, semaphore does not
- mutex is a lock mechanism, semaphore is a signaling mechanism
  please read `https://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_mutex_lock.html`

A2. MMX / SSE / AVX

These are all SIMD programming. SIMD stands for single instruction, multiple data. Here is a comparison :

```
        | register size | header
--------+---------------+--------------
1. MMX  |  64 bits      |  mmintrin.h
2. SSE  | 128 bits      | xmmintrin.h
   SSE2 | 128 bits      | emmintrin.h
   SSE3 | 128 bits      | pmmintrin.h
   SSE4 | 128 bits      | nmmintrin.h == 16 char / 8 std::uint16_t / 4 std::uint32_t / 2 std::uint64_t / 4 float / 2 double
3. AVX  | 256 bits      | immintrin.h
```

Lets have a simple test. We need to copy data into specific register (128 bits for SSE) and copy the result back. It incurs costs.

```
    #include <iostream>
    #include <math.h>
    #include <time.h>
    #include <nmmintrin.h>
    #include <immintrin.h>

    __m128 a = _mm_set_ps(1.0f, 1.0f, 1.0f, 1.0f);
    __m128 b = _mm_set_ps(1.0f, 2.0f, 3.0f, 4.0f);
    __m128 c = _mm_add_ps(a,b);
    __m128 d = _mm_sub_ps(a,b);
    __m128 e = _mm_mul_ps(a,b);
    __m128 f = _mm_div_ps(a,b);
    __m128 g = _mm_rcp_ps(b);   // reciprocol
    __m128 h = _mm_sqrt_ps(b);  // square root

    // Byte order is reversed
    std::cout << "\nresult c0 = " << reinterpret_cast<float*>(&c)[3];
    std::cout << "\nresult c1 = " << reinterpret_cast<float*>(&c)[2];
    std::cout << "\nresult c2 = " << reinterpret_cast<float*>(&c)[1];
    std::cout << "\nresult c3 = " << reinterpret_cast<float*>(&c)[0];

    std::cout << "\nresult d0 = " << reinterpret_cast<float*>(&d)[3];
    std::cout << "\nresult d1 = " << reinterpret_cast<float*>(&d)[2];
    std::cout << "\nresult d2 = " << reinterpret_cast<float*>(&d)[1];
    std::cout << "\nresult d3 = " << reinterpret_cast<float*>(&d)[0];
    // and so on ...
```

B*i*. Cache and cache hierarchy

What is cache?

CPUs operate considerably faster than main memory *RAM*. *CPUs* usually find themselves ***starved for data*** and become idled while waiting for data from main memory. Hence memory access is the bottleneck for system performance, a lot of effort is put in making memory access faster, such as : cache and *NUMA*.

Cache hierarchy

Cache hierarchy (memory spectrum) in increasing latency and increasing number

- register *1 cycle*
- L1 cache *4 cycles*
- L2 cache *10 cycles*
- L3 cache *50 cycles*
- primary storage : *DRAM* *200 cycles*
- secondary storage : harddisk *2ms* access time copied from *"Dreamrunner Low latency programming"*

B*ii*. Cache hit/miss and locality of reference

Cache hit/miss

Cache is a copy of main memory data for *later-quick-retrieval*. When *CPU* attempts to accesses an object with specific address, it will firstly look for it in L1 cache, if it is found, retrieval is done, this is called cache-hit. On the contrary if cache-miss happens, *CPU* will continue the search by escalating the cache-hierarchy through L2/L3 cache and up to main memory for the object. Once the object is finally found, the ***object*** and ***its neighbourhood*** are copied to caches for *later-quick-retrieval*. Basic idea of caching mechanism is that we can keep a small set of frequently-accessed variables in the expensive cache, while leaving the rest infrequently-accessed data in the slow main memory. The question is … why both the ***object*** and ***its neighbourhood*** are cached? Temporal and Spatial locality.

Locality of reference

Temporal locality means a memory location currently-accessed is likely, *with a high prior probability*, to be accessed again in the near future, thus currenty-accessed ***object*** is cached. ***Spatial locality*** means the neighbourhood of a memory location currently-accessed is likely to be accessed next, as correlated memory locations usually sit next to each other (belong to same object or same container). In other words, caching mechanism assumes users programmes exhibit a pattern that follows temporal locality and spatial locality, if its speculation is right, system performance can be optimized as most memory accesses are cache-hit. However if we do not write cache-friendly code, violating the locality-of-reference assumptions, resulting in frequent cache-miss, then system performance will be significantly degraded, as *CPU* is idled waiting for data.

B*iii*. Cache line and two cache associativities

Cache line

Cache is partitioned into units called ***cache line***, which is usually *64 bytes* in size, each cache line is a copy of a *64 bytes* block in main memory with starting address and ending address divisible by *64*, which can be found by truncating the last *6 bits* of the address of the to-be-cached object. For example if an object with *32-bit* address `ABCD0012` is cached, the whole block from `ABCD0000` to `ABCD003F` is cached into a single cache line. Cache is limited, Intel® Core™ i7 Processor has *32K bytes* L1 cache, or equivalently, *512* cache lines. Each cache line is identified by an index, says *0-511*. Each cache line is shared among multiple blocks in main memory.

main memory	*cache*
memory size = 2^{32}	*cache size* =32*K* = 2^{15}
num of blocks = $2^{32}/2^6 = 2^{26}$	*num of blocks* = $2^{15}/2^6 = 2^9 = 512$

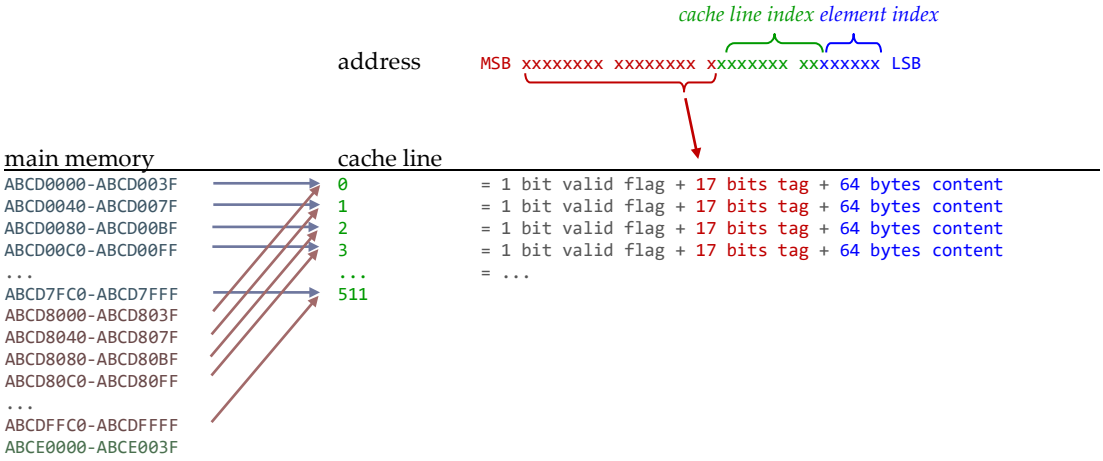
num of blocks sharing same line = $2^{26}/2^9 = 2^{17}$

There are different schemes for mapping ***block-address*** to ***cache-index***, called cache associativity. Two common examples :

- direct-map
- set-associativity

Direct-map

Direct-map is a simple many-to-one deterministic cache associativity. Blocks are sequentially assigned to cache line `0,1,2,3,....,511` (suppose there are *512* cache lines) and then rollbacks to `0` again, hence cache index rollbacks once every 2^{9+6} memory locations, and we can derive cache index simply by truncating the first *17 bits (32-9-6 bits)* and the last *6 bits* of block-address. In the cache circuitry, apart from the *64 bytes* content, each cache line should contain one ***valid-bit*** and one ***tag***, which represent whether it is a valid copy of main memory and the exact location of memory block. Using the previous example, the first *17 bits* form the tag, while the next *9 bits* form the cache index, and the last *6 bits* are don't care.



Disadvantage of direct-map is that, when two uncorrelated memory blocks being mapped to the same cache line, keep accessed by single thread (or multithreads, it doesn't matter) continuously, the two blocks keep expelling each other from the cache by copying itself, resulting in frequent cache-miss. This problem is called ***line sharing***. Line sharing does not involve multithreading, and hence is unrelated to ***cache-coherence*** mechanism, like *MESI*, across different *CPU* cores, it is **not** about ***cache-ping-pong*** neither. Solution to ***line sharing*** is to apply a more dynamic cache associativity which does not limit a memory block to single specific cache line. Typical associativity include : set-associativity and full-associativity, the latter is just an extreme of the former.

Set-associativity

Consider a 4-way set-associative cache, it groups *4* cache lines into *1* cache set, the original *128K*×*4* memory blocks can share the set in a first-come-first-serve manner. When the set is full, the least-recently-used replacement algorithm then kicks in. Any 2^N way set-associative cache can be defined in the same manner. With set-associativity, the probability of ***line sharing*** is reduced.

B*iv*. Cache coherency

Cache coherency mechanism

Cache is an image (snapshot) of selected locations of main memory :

- for multi core system, cache of the cores capture selected locations in memory at different time instances
- for multi core system, cache coherency mechanism (such as MESI protocol) is needed for synchronization
- MESI protocol defines 4 cache-line states : `modified`, `exclusive`, `shared` and `invalid` (for each cache line in each core)
- MESI protocol defines MESI messages between cores : `invalidate-msg`, `invalidate-ack` and many other

When one cache line in main memory being cached in more than one cores :

- when the cache line is synchronized among those cores, then the cache line has a `shared` state in all the cores
- when the cache line is modified by one of the cores, hand shaking of `invalidate-msg` and `invalidate-ack` happens among cores
- cache line states are changed to `modified` for the core initiating the change and `invalid` for the core notified on the change
- hand shaking is blocking, the cores need to wait for acknowledgement before triggering cache-line state change

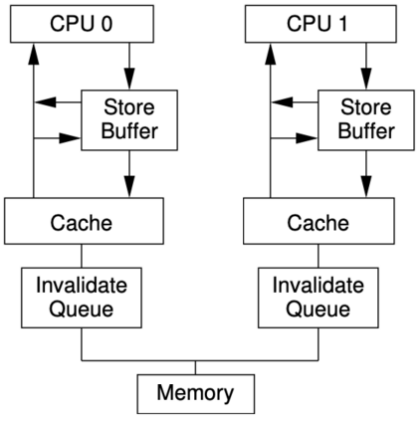
Store buffer and invalidate queue

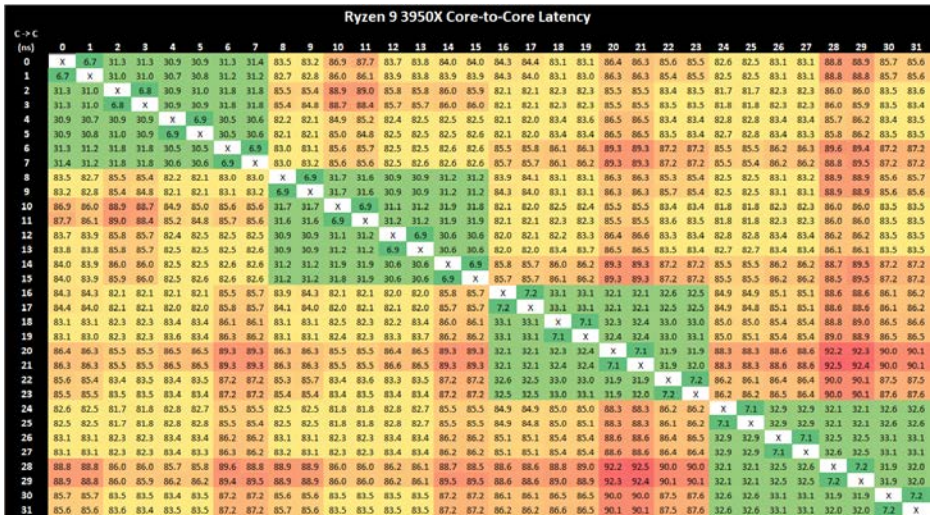
This blocking hand shaking mechanism is bad for low latency :

- it can be avoided by introducting store buffer and invalidate queue for each core
- consider the core initiating a change, instead of invoking `blocking_send(invalidate-msg)` ...
- the core invokes `non_blocking_send(invalidate-msg)`, inserts `modified` state into store buffer, continues to execute next instruction
- the `modified` state will be popped from store buffer and becomes effective when `invalidate-ack` is received later

Use of store-buffer and invalidate-queue introduces a new problem, the sequence inconsistency in multi-threading scenario :

- sequence inconsistency is common in publication pattern under multithreading scenario
- as publication and ready-flag are both cached among the cores
- as publication and ready-flag may have different cache-states among the cores
- as a result, publication may be synchronized before ready-flag





Memory order comes to rescue

The problem can be solved by adding memory order, which are constraints that flushes store buffer and invalidate queue.

- fence `memory_order_release` forces a core to handle store buffer immediately
- fence `memory_order_acquire` forces a core to handle invalidate queue immediately
- hence `memory_order_release` is equivalent to forbidding preceding stores from being **moved past** current store
- hence `memory_order_acquire` is equivalent to forbidding subsequent loads from being **moved before** current load

Please search image on web : *inter core data latency*. Due to different *CPU* structure on the chip, inter core latency varies.

B1. Loop-blocking technique

Non cache friendly implementation

A typical example to demonstrate loop blocking technique is matrix multiplication, which involves 3-layer nested for-loop.

```
template<typename T, unsigned N> struct matrix
{
    const T& operator()(unsigned y, unsigned x) const { return values[y*N+x]; }
          T& operator()(unsigned y, unsigned x)       { return values[y*N+x]; }

    alignas(64) T values[N*N];
};

template<typename T, unsigned N> void multiply(const matrix<T,N>& A, const matrix<T,N>& B, matrix<T,N>& C)
{
    for(unsigned y=0; y!=N; ++y)
    {   for(unsignd x=0; x!=N; ++x) // line 0
        {
            C(y,x) = 0; // line 1
            for(unsigned z=0; z!=N; ++z) C(y,x) += A(y,z)*B(z,x); // line 2
        }
    }
}
```

Given a machine with only *3* cache lines, each line is *64 bytes* large, we run the above naïve implementation with `T` as *4 bytes* `int` and `N=64`, then cache-hit rate of the innermost loop in `line 2` is `60/(64×2) = 60/128` :

```
C(y,x) = A(y, 0)*B( 0,x) + A(y, 1)*B( 1,x) + A(y, 2)*B( 2,x) + ... + A(y,15)*B(15,x) +
         miss    miss      hit     miss      hit     miss            hit     miss
         A(y,16)*B(16,x) + A(y,17)*B(17,x) + A(y,18)*B(18,x) + ... + A(y,31)*B(31,x) +
         miss    miss      hit     miss      hit     miss            hit     miss
         A(y,32)*B(32,x) + A(y,33)*B(33,x) + A(y,34)*B(34,x) + ... + A(y,47)*B(47,x) +
         miss    miss      hit     miss      hit     miss            hit     miss
         A(y,48)*B(48,x) + A(y,49)*B(49,x) + A(y,50)*B(50,x) + ... + A(y,63)*B(63,x) +
         miss    miss      hit     miss      hit     miss            hit     miss
```

- assume the *CPU* is smart enough to assign one cache line to one matrix, each line can carry *16* `int`
- `alignas(64)` assigns the starting address of `values[N*N]`to multiple of `64`
- all *15* subsequent accesses of matrix `A` after `A(y,0)` are cache-hit as they are all cached when `A(y,0)` is invoked
- all *15* subsequent accesses of matrix `B` after `B(0,x)` are cache-miss
- no caching occurs when `A(y,1) A(y,2) ...` as they are cache-hit
- caching occurs after each cache-miss
- caching for matrix `B` is wasted
- we can push cache-hit rate to near *100%* by re-arranging the for-loops :
- in order to make use of `B` cache, we handle *16* entries in `C` simultaneously instead of one entry each time, hence we
- modify `line 0` so that `x` is incremented by `x+=16` instead of `++x`
- modify `line 1` in a loop that fills *16* zeros
- modify `line 2` in a loop that scales *16* entries

To facilitate loop block, lets introduce a vector view of the matrix.

```
template<typename T, unsigned N> struct vector
{
    vector(matrix<T,N>& m_, unsigned y_, unsigned x_, unsigned size_ = 1) : m(m_), y(y_), x(x_), size(size_) {}

    void set(T value)
    {
        for(unsigned offset=0; offset!=size; ++offset) m(y,x+offset) = value;
    }

    void add(T scale, vector<T,N>& rhs)
    {
        for(unsigned offset=0; offset!=size; ++offset) m(y,x+offset) += scale * rhs.m(rhs.y,rhs.x+offset);
    }

    matrix<T,N>& m;
    unsigned y;
    unsigned x;
    unsigned size;
};
```

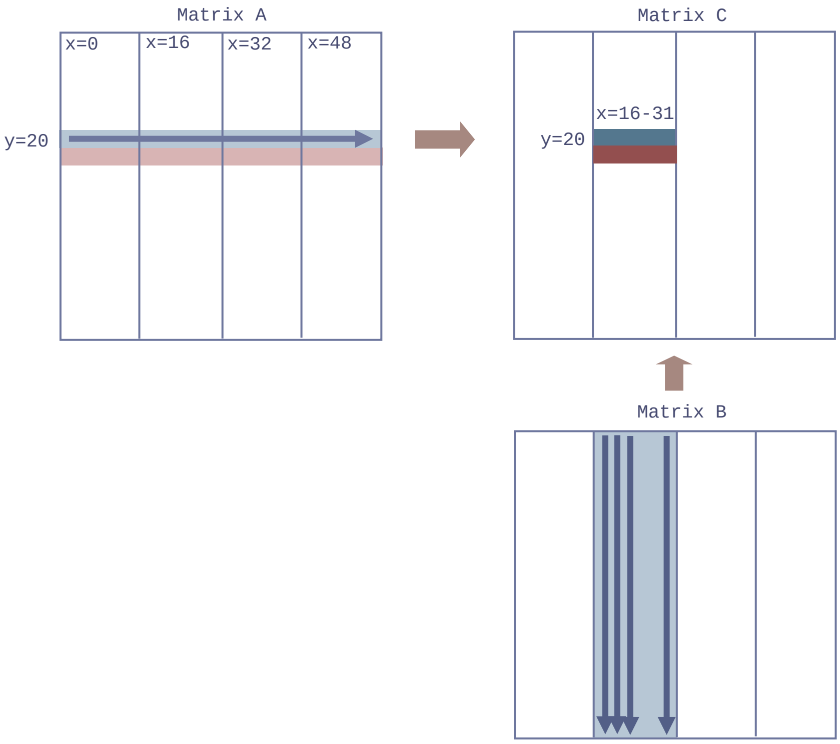
Cache friendly implementation

Here is the cache friendly version, with `line 0/1/2` replaced by `line 3/4/5`.

```
// We should have declared :
// const matrix<T,N>& A and
// const matrix<T,N>& B
// but doing so, we need to offer both vector and const_vector, hence omitted for simplicity.

template<typename T, unsigned N> void multiply(matrix<T,N>& A, matrix<T,N>& B, matrix<T,N>& C, unsigned cache)
{
    for(unsigned y=0; y!=N; ++y)
    {
        for(unsigned x=0; x!=N; x+=cache) // line 3
        {
            vector<T,N>{C,y,x,cache}.set(0); // line 4
            for(unsigned z=0; z!=N; ++z) vector<T,N>{C,y,x,cache}.add(A(y,z), vector<T,N>{B,z,x,cache}); // line 5
        }
    }
}
```

It is like partitioning the matrix into vectors, each vector is `1×16` large. We then iterate through the ***matrix of vectors***, for each vector, initialize it to all-zero, and accumulate the product between `A` and `B`. The following shows what happen when `y=20` and `x=16:31`.



Lets calculate cache-hit rate of the innermost loop in `line 5` again. For fair comparison, we ignore caching for matrix `C`.

```
                                                                                            cached stuffs
C(y,x+0)+= A(y,0)*B(0,x+0);    C(y,x+1)+= A(y,0)*B(0,x+1);    ... C(y,x+15)+= A(y,0)*B(0,x+15);    A(y,0:15)  B(y,0:15)
           miss   miss                    hit    hit                           hit    hit
C(y,x+0)+= A(y,1)*B(1,x+0);    C(y,x+1)+= A(y,1)*B(1,x+1);    ... C(y,x+15)+= A(y,1)*B(1,x+15);    A(y,0:15)  B(y,0:15)
           hit    miss                    hit    hit                           hit    hit
C(y,x+0)+= A(y,2)*B(2,x+0);    C(y,x+1)+= A(y,2)*B(2,x+1);    ... C(y,x+15)+= A(y,2)*B(2,x+15);    A(y,0:15)  B(y,0:15)
           hit    miss                    hit    hit                           hit    hit
...
C(y,x+0)+= A(y,16)*B(16,x+0); C(y,x+1)+= A(y,16)*B(16,x+1); ... C(y,x+15)+= A(y,16)*B(16,x+15);   A(y,16:31) B(y,0:15)
           miss    miss                   hit     hit                          hit     hit
...
C(y,x+0)+= A(y,63)*B(63,x+0); C(y,x+1)+= A(y,63)*B(63,x+1); ... C(y,x+15)+= A(y,63)*B(63,x+15);   A(y,48:63) B(y,0:15)
           hit     miss                   hit     hit                          hit     hit
```

There are $4+64$ cache-miss within $64\times16\times2$ cache read, thus :

```
cache-hit-rate = (64×16×2-4-64)/(64×16×2) ~ 100%
```

B2. Instruction cache-miss

A junction operator heard a train coming, he had no idea which way it might go, he could either :
- *stop the train and ask the driver, then set the junction switch appropriately, or*
- *make a (prior probability) guess, there are two outcomes :*
- *if the guess is right, the train will move on*
- *if the guess is wrong, the driver will stop / back up / yell to flip the switch*

Providing that the train is heavy, it takes forever to startup and slow down and
- *if there exists a high probability of right guess*
- *then the latter is a better approach. This is what CPU pipeline does.*

Pipeline and branch prediction

In modern processors, instruction execution is divided into many substeps and run by long pipelines. It takes time to warm up and slow down a running pipeline, like a heavy train. When processor comes across a branch in the code, it performs ***branch prediction*** based on ***historical pattern*** and moves on. In case of wrong prediction, prefetched instructions are useless, this is called ***instruction cache-miss***, the processor will :
- flush the pipeline
- roll back to branching point
- restart down the other path

If the guess is right everytime, the execution will never have to stop. For low latency, developer should minimise ***instruction cache-miss*** by making well-behaved branch (i.e. predictable branch).

Avoid unpredictable branches by sorting

Given an array of random `unsigned char`, count the number of items above threshold `128`. The following example has unpredictable branch which results in frequent instruction cache-miss. It can be solved by sorting input vector first.

```
// Example 1
int count(const std::vector<unsigned char>& vec)
{
     int ans = 0;
     for(int n=0; n!=vec.size(); ++n) if (vec[n] >= 128) ++ans;
     return ans;
}

std::vector<unsigned char> vec;
for (int x=0; x!=100; ++x) vec.push_back(rand()%256);
std::cout << count(vec);
```

Avoid unpredictable branches by remove IF

It can also be solved by replacing *IF-logic* by *non-IF-logic*, such as RHS *bit-shift* operations, sacrificing readability.

```
// Example 1
int count(const std::vector<unsigned char>& vec)
{
     int ans = 0;
     for (int n=0; n!=vec.size(); ++n) ans += (vec[n]>>7);
     return ans;
}

// Example 2 - Dont write this
bool inside_range(double x, double y)                  assembly code :    if not fulfill x < LHs, jump 2 lines
{                                                                         return false
     if (x < LHs)   return false;                                         if not fulfill x > RHS, jump 2 lines
     if (x > RHs)   return false;                                         return false
     if (y < lower) return false;                                         ...
     if (y > upper) return false;                                         For true case, it involves 4 jumps.
}

// Example 2 - Write this. NO JUMP at all
bool inside_range(double x, double y)
{
     return (x > LHs && x < RHS && y > lower && y < upper);
}
```

Avoid virtual functions

Existence of virtual functions degrades programme performance for *3* reasons, the last one is dominant and cache-related :

- virtual function involves two dereferencing for every invocation : pointer to `vtable` and pointer to member function
- virtual function makes a class no longer POD, POD optimizations, such as `memcpy`, do not work anymore
- virtual function leads to unpredictable branch and hence instruction cache-miss
- pipeline cannot prefetch instructions until the virtual function is resolved

Using built in functions

Compile `gcc` offers a lot of built-in functions, which are `gcc` specific and not included in standard C++, by making use of knowledge about the compiler itself. Some of them are low-latency related, such as :

- hint for branch prediction
- pause CPU for several cycles inside `while(true)` loop, such as spinlock *(not related to branch prediction)*
- compile option to forbid built-in `memcmp` *(not related to branch prediction)*
- ► read `https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html` for whole list

Hint for branch prediction

Given a if-else-block, if the probabilities of firing various branches differ significantly, then we can give the compiler a hint so as to facilitate branch prediction. We can tell if there is a high probability for an expression to be `true`, or not by :

```
// The following hint tells branch prediction to give fctA always (or with a higher prob).
if (__buildin_expect(expression, true)) fctA();
else fctB();

// or equivalently, use [[likely]]                  In assembly, compiler puts fctA in next line, while fctB involves a short-jump.
if (expression) [[likely]] fctA();
else fctB();


// The following hint tells branch prediction to give fctD always (or with a higher prob).
if (__buildin_expect(expression, false)) fctC();
else fctD();

// or equivalently, use [[unlikely]]
if (expression) [[unlikely]] fctC();
else fctD();


// Common use case :
if (__buildin_expect(ptr!=nullptr, true)) ptr->fct();

// Common use case :
if (ptr!=nullptr) [[likely]] ptr->fct();
```

Pause CPU for several cycle inside `while(true)` loop (NOT related to branch prediction)

We usually implement spinlock with a `while(true)` loop and we do nothing inside the loop, as a result, due to unknown reasons, the thread releasing the spinlock is likely to get the spinlock again, in a race for that spinlock against other threads, so that thread ends up consuming CPU resources, leading to starvation for other threads. A solution is to pause the thread for several cycle inside loop. Another solution is to call `std::this_thread::yield()`, however it may result in poor latency.

```
// pause current thread for several clock cycle
while(flag.test_and_set(std::memory_order_acquire)) __builtin_ia32_pause();

// equivalent to
while(flag.test_and_set(std::memory_order_acquire)) asm("pause");

// also equivalent to
while(flag.test_and_set(std::memory_order_acquire)) asm("no-op");
```

Compile option to forbid built-in `memcmp` (NOT related to branch prediction)

`gcc` compiler provides a built-in version `memcmp`, however this version is slower than the one offered by standard C++. In this case, we want to disable `gcc` version `memcmp`. It can be done by compile option, for example, in `cmake` :

```
add_compile_options(-MMD -MD -Wall -Wextra -Wpedantic -Werror -fno-builtin-memcmp -pthread)
```

B3. Cache line sharing

Three cache line sharing

Line sharing is **not** about concurrency, it does happen in single thread, it is the result of sharing one cache line by multiple memory blocks. Both true sharing and false sharing involves multithreading. True sharing means multiple threads accessing the same object, thus accessing the same memory location and the same cache line. False sharing means multiple threads accessing different objects, which unfortunately lie in the same memory block and hence mapped to the same cache line. In both true sharing and false sharing if the threads belong to different *CPUs* or different cores in the same *CPU*, ***cache coherence mechanism*** then kicks in to synchronize the content in main memory and cache of different cores. One common ***cache coherence mechanism*** is the ***MESI protocol***, it defines 4 possible states for each cache *(modified, exclusive, shared, invalid)* and a set of possible state-transitions. When multiple cores having one cache line mapped to the same memory block, and when one memory location in that cache line is modified by one of the cores, the corresponding cache line of all the other cores must be invalidated through MESI protocol, which has to be read again from the main memory the next time it is needed. If multiple cores modify that cache line in turn frequently, the cache coherence mechanism will experience massive slowdown because of rapid successive cache line invalidation and data shuttled from main memory.

There are three types of cache line sharing :

1	*line sharing*	*same cache line shared by multi mem blocks, happen in single thread (no cache coherency, no pingpong)*
2	*true sharing*	*same cache line, and same main mem location, is shared by multithreads (pingpong)*
3	*false sharing*	*same cache line, but multi main mem locations, is shared by multithreads (pingpong)*

Solution to line sharing

See cache associativity.

Solution to true sharing

This is known as ***cache ping-pong***. Cache ping-pong due to true sharing can be easily spotted out by code review and thus avoided. However cache ping-pong due to false sharing is an easy mistake made unintentionally and difficult to detect. False sharing can be avoided by placing variables in different memory blocks, if they will be accessed by different threads. This can be done by padding (by adding dummy spaces between variables of interest) or manual alignment (by adding `alignas` before declaring variables) or by different division-of-labour among threads.

3 solutions to false sharing

A typical example of false sharing is the `next_write` and `next_read` indices in container used in producer consumer model. Producer thread modifies `next_write` while consumer thread modifies `next_read` independently, developer may place them side by side in class definition unintentionally, the two indices are mapped to the same cache line, resulting in cache ping-pong. Similarly, the container usually contains a raw array of ticks, consecutive ticks may share the same cache line, and trigger false sharing. Techniques :

- add padding to members, such as `tick::dummy`
- add alignment to members, such as `next_write` and `next_read`

```
struct padding { char x[64]; };
struct tick // try to keep it POD, so that memcpy works
{
    order bid_queue[MAX_QUEUE_SIZE];
    order ask_queue[MAX_QUEUE_SIZE];
    transaction last_trade;
    padding dummy;
};

class timeseries
{
    void get_functions();
    void set_functions();
    alignas(std::hardware_destructive_interference_size) unsigned long next_write = 0;
    alignas(std::hardware_destructive_interference_size) unsigned long next_read = 0;
    tick ticks[65536];
};
```

- If we want to process a vector with two threads, if there are two possible division-of-labour schemes :

- `threadA` processes odd items, while `threadB` processes even items or
- `threadA` processes first half, while `threadA` processes second half → this is definitly a better choice

where ...

`std::hardware_destructive_interference_size`	minimum offset between two objects to avoid false sharing
`std::hardware_constructive_interference_size`	maximum offset between two objects to promote cache hit

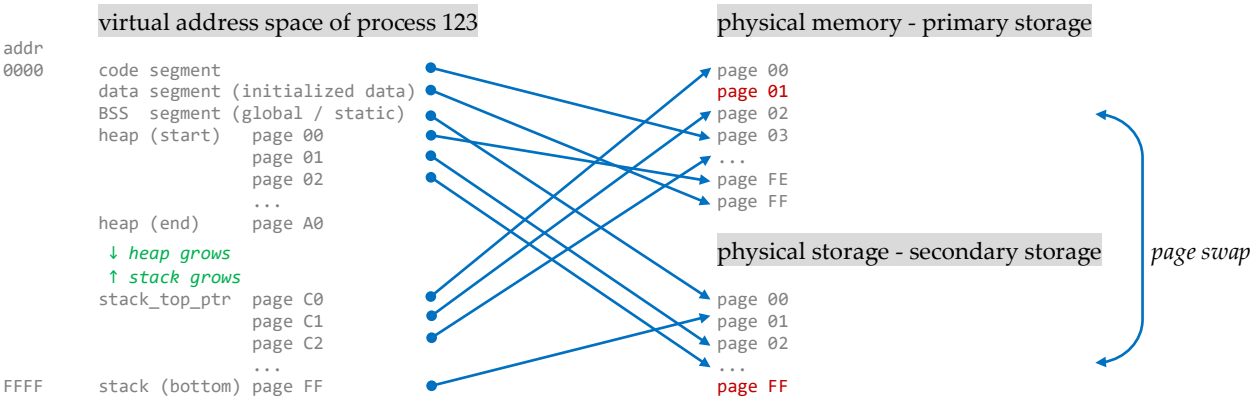
B4. Avoid paging and thrashing

Virtual address space and paging

Multiprocessing *OS* needs to share primary memory (*RAM*) among multiple processes effectively, so as to maximize the number of processes running concurrently in primary memory while avoid hitting the performance of processes. Primary memory is a limited resource, at the same time, it is not necessary to put the whole process in primary memory at any given time, which means that we can put part of it in secondary storage (hard disk), so here comes the virtual address space (*VAS*) concept. Physical memory can be classified into primary storage and secondary storage, both of them are partitioned into pages with fixed size. Each process is then assigned with *VAS*, which is a set of pointers to the pages in physical memory, managed by a virtual memory manager *(VMM)*.

The pages assigned to a process are thus :

- pages are not contiguous physically
- current accessed pages are in primary storage
- least recently used pages are in secondary storage



How does paging work?

- when the process accesses a variable in a page that exists in primary page, everything is fine ...
- when the process accesses a variable in a page that does not exist in primary page, ***page fault*** is generated, ***paging*** kicks in :
  - `paging out` = *VMM* moves a least-recently-used *(LRU)* page from *RAM* to harddisk as a ***paging file***
  - `paging in` = *VMM* moves the requested ***paging file*** from harddisk to *RAM*
- this swapping process is called ***paging***, this process involves *swaping the whole page* and *updating pointers*

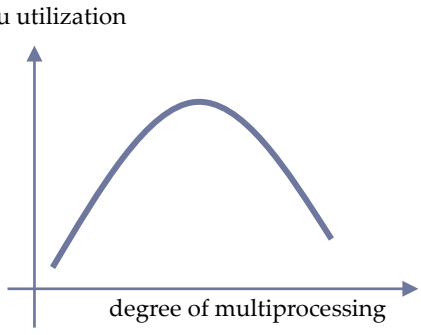
Do not confuse ***page*** with ***memory block*** :

- ***page*** is memory model among different processes, used in *VAS*, invoke paging on page fault
- ***memory block*** is memory model for cache (*64 bytes* in size), move data from *RAM* to cache on cache miss
- page can be considered as a part of the cache hierarchy, page fault is thus a kind of cache miss

Paging and thrashing

Paging is a compromising mechanism between maximizing number of concurrent processes and process performance, however for time critical low latency application, it is a bad thing. What even worse is the possibility of thrashing, which is a self-reinforcement loop of page-fault and paging, leading to significant degrade, or collapse in system performance. How does thrashing happen? The set of pages frequently accessed by a program is called ***working set***. When working set of a program is large that cannot fit into the primary storage, then :

- page fault occurs and paging starts
- *CPU* becomes idle waiting for paging to complete
- *CPU* utility is low and *OS* tends to increase *CPU* efficiency by starting more processes
- other processes may result in more page faults



How to detect thrashing?

- using *CPU* profiler, it becomes unreasonably low
- using virtual memory status command `vmstat`

Avoid paging and thrashing by ***memory lock***

We can forbid a piece of memory from paging out using function `mlock()`, we have to specify the beginning address and the length of that piece of memory. However the beginning address should be multiples of page size, hence we need to get the page size using function `sysconf()`, adjust the offset, while keeping the end unchanged, length has to be adjusted too. Reverse is done by `munlock()`.

```
// This is called memory-lock. Don't confuse it with memory-barrier.
#include <unistd.h>
#include <sys/mman.h> // for mlock() and munlock()

void lock_memory(char *addr, size_t size)
{
     std::uint64_t page_size   = sysconf(_SC_PAGE_SIZE); // standard function to get page size
     std::uint64_t page_offset = (std::uint64_t)addr % page_size;
     addr -= page_offset;
     size += page_offset;
     return mlock(addr,size); // lock page in stack memory
}

void unlock_memory(char *addr, size_t size)
{
     std::uint64_t page_size   = sysconf(_SC_PAGE_SIZE); // standard function to get page size
     std::uint64_t page_offset = (std::uint64_t)addr % page_size;
     addr -= page_offset;
     size += page_offset;
     return munlock(addr,size); // unlock locked page
}

// The page containing object will be forbidden from paging-out.
T frequently_used_object;

lock_memory(&frequently_used_object, sizeof(T));
time_critical_part();
unlock_memory(&frequently_used_object, sizeof(T));
```

Besides, we can also lock every pages of the process by `mlockall()`, which takes a input flag. That flag can be :

- `MCL_CURRENT` for locking all existing pages for the process before calling `mlockall(MCL_CURRENT)`
- `MCL_FUTURE` for locking all growing pages for the process after calling `mlockall(MCL_FUTURE)`

What are existing pages and growing pages? Please take a look at the snapshot of the VAS in previous page :

- heap grows as we do further `malloc()`
- stack grows as we call more functions

Besides, there is an upper limit on the size of memory locking which can be observed by command `ulimit -a` and modified by `ulimit -l unlimited`. If we invoke `mlockall(MCL_FUTURE)`, then future invocation of `malloc()` may fail because of reaching that limit. We have to either check the return value from `malloc()` or to set a larger limit using `ulimit`.

Reference

What is memory locking?

https://www3.physnet.uni-hamburg.de/physnet/Tru64-Unix/HTML/APS33DTE/DOCU_005.HTM

C2. Context switch

What is context switch?

In a concurrent system, we can run more threads than physical *CPUs* or cores, through time-interleaving managed by the scheduler. A thread must be switched out of a *CPU* so another thread can run, the following are done :

- whole set of states of original thread has to be saved, so that its states can be restored when the thread resumes
- whole set of states of new thread has to be load, so that it can be run

This process is called context switch, which is time consuming. We can see context switch by :

```
std::mutex io_mutex;
std::vector<std::thread> threads(4);
for(int n=0; n!=4; ++n)
{
     threads[i] = std::thread([&io_mutex,n]
     {
          while(true)
          {
              {
                   std::lock_guard<std::mutex> io_lock(io_mutex);
                   std::cout << "Thread" << n << " on CPU " << sched_getcpu(); // sched_getcpu() in linux only?
               }
               std::this_thread::sleep_for(std::chrono::seconds(1));
          }
    });
}
for(auto& t:threads) t.join();
```

```
Thread #0: on CPU 5
Thread #1: on CPU 5
Thread #2: on CPU 2
Thread #3: on CPU 5
Thread #0: on CPU 2
Thread #1: on CPU 5
Thread #2: on CPU 3
Thread #3: on CPU 5
Thread #0: on CPU 3
Thread #2: on CPU 7
Thread #1: on CPU 5
Thread #3: on CPU 0
Thread #0: on CPU 3
Thread #2: on CPU 7
Thread #1: on CPU 5
Thread #3: on CPU 0 ^C
```

3 solutions

In order to avoid context switch, we have to :

- pin one **thread** to one *CPU* by setting affinity
- dedicate one *CPU* to one **process** by setting nice value / priority / scheduler policy
- isolate one *CPU* from other **processes** by setting in BIOS parameter `isolcpus`

Linux - nice value vs priority

Priority of a process / a thread in linux is complicated, it consisted of :

- ***nice value*** (`NV`) which is a hint for OS to prioritize processes / threads — `NV∈[-20,19]` where `-20` is the highest
- ***priority*** (`PR`) which is a rank for OS to prioritize processes / threads — `PR∈[-100,-2]∪[0,39]` where `-100` is the highest
- ***scheduler policy*** which is an algorithm that governs prioritization, default policy is `SCHED_OTHER`

- three normal policies : `SCHED_IDLE` (the lowest priority), `SCHED_BATCH` (run as batch) and `SCHED_OTHER` (run as round robin)
- two real-time policies : `SCHED_FIFO` (FIFO real time mode) and `SCHED_RR` (round robin real time mode)
- policy `SCHED_FIFO` is the highest among all, it forces scheduler to run current process unless it calls `yield` or `sleep` only

We can view `NV` and `PR` by `top` command. The 3 concepts work together like below :

```
if (is_realtime_policy(current_policy))
     PR = -1 - realtime_priority;
else PR = 20 - NV;
```

with `realtime_priority∈[1,99]` where `99` is the highest
with `NV∈[-20,19]` where `-20` is the highest
hence `PR∈[-100,-2]` for realtime policy, where `-100` is the highest
hence `PR∈[0,39]` for normal policy, where `0` is the highest

Nice value is just a hint for scheduler, there is no guarantee (very often no use). Priority is a stronger setting.

Affinity and priority in windows

This is thread-affinity and thread-priority in `cubquant` for windows using STL (note both are for thread) :

```
auto num_logical_cpus = std::thread::hardware_concurrency();

void set_this_thread_affinity(int affinity)
{
     DWORD_PTR mask=0x1;
     for(int n=0; n!=affinity; ++n) mask = mask << 1;
     SetThreadAffinityMask(GetCurrentThread(), mask); // for thread
}

void set_this_thread_priority(int priority)
{
     SetThreadPriority(GetCurrentThread(), priority); // for thread
}
```

Affinity and nice-value / priority / scheduler-policy in linux

This is thread-affinity in linux :

```
void set_this_thread_affinity(int affinity)
{
     cpu_set_t cpuset;
     CPU_ZERO(&cpuset);            // set zero
     CPU_SET(affinity, &cpuset);   // set desired value
     pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset); // for thread
}
```

This is nice-value for a process in linux :

```
#include <sys/resource.h>
void set_this_process_nice_value(int nice_value = -20)
{
     setpriority(PRIO_PROCESS, getpid(), nice_value); // for process
}
```

Priority and scheduler policy are set together with a single function in linux. It can be done in process level or in thread level. In the following demonstration, we firstly get the maximum real-time priority of a specific policy into `sched_param`, then input both desired policy and `sched_param` into function `sched_setscheduler` (for process level) or `pthread_setschedparam` (for thread level).

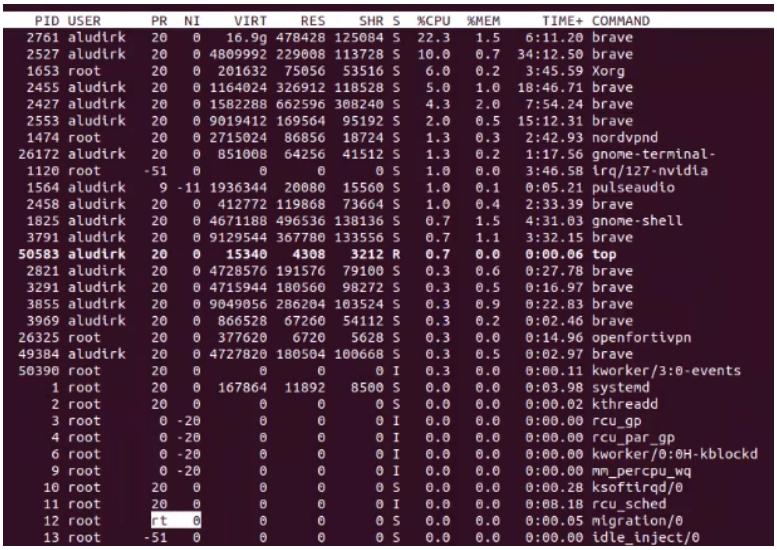
```
#include <sched.h>
bool set_this_process_priority(auto policy) // where policy SCHED_FIFO, SCHED_RR, SCHED_OTHER, SCHED_BATCH, SCHED_IDLE
{
     struct sched_param para;
     para.sched_priority = sched_get_priority_max(policy); // this is also called realtime-priority
     if (sched_setscheduler(getpid(), policy, &para) < 0) return false;
     return true;
}

bool set_this_thread_priority(auto policy)
{
     struct sched_param para;
     para.sched_priority = sched_get_priority_max(policy); // this is also called realtime-priority
     if (pthread_setschedparam(std::this_thread::native_handle(), policy, &para) < 0) return false;
     return true;
}
```

Please read website `superuser`, question `203657` :

- we can apply both policy and real-time priority together
- we can apply `SCHED_FIFO` policy on process level or on thread level
- if we apply `SCHED_FIFO` policy on process level and if its affinity is set, then :
  - all threads spawned from this process will run in the same core with `SCHED_FIFO` policy, resulting in ***system hang***
  - solution is ***NOT*** to spawn threads in this process or
  - solution is to apply `SCHED_FIFO` policy on thread level
- if we apply `SCHED_FIFO` policy on thread level for multiple threads, having affinity pinned on different core
  - remember to setting affinity before setting `SCHED_FIFO` (i.e. order does matter), otherwise if we do the other way round ...
  - multiple threads are assigned to random cores (at least before affinity is set), if
  - multiple threads are assigned to the same core, while running with `SCHED_FIFO`, again result in ***system hang***
- running function `sched_get_priority_max` require superuser, hence `sudo` is needed to run the executable

After running as real time process, you can verify it by `top` command, and read columns `PR(priority)=RT(realtime) NICE=-20`



```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 2761 aludirk   20   0   16.9g 478428 125084 S  22.3   1.5   6:11.20 brave
 2527 aludirk   20   0 4809992 229008 113728 S  10.0   0.7  34:12.50 brave
 1653 root      20   0  201632  75056  53516 S   6.0   0.2   3:45.59 Xorg
 2455 aludirk   20   0 1164024 326912 118528 S   5.0   1.0  18:46.71 brave
 2427 aludirk   20   0 1582288 662596 308240 S   4.3   2.0   7:54.24 brave
 2553 aludirk   20   0 9019412 169564  95192 S   2.0   0.5  15:12.31 brave
 1474 root      20   0 2715024  86856  18724 S   1.3   0.3   2:42.93 nordvpnd
26172 aludirk   20   0  851008  64256  41512 S   1.3   0.2   1:17.56 gnome-terminal-
 1120 root     -51   0       0      0      0 S   1.0   0.0   3:46.58 irq/127-nvidia
 1564 aludirk    9 -11 1936344  20080  15560 S   1.0   0.1   0:05.21 pulseaudio
 2458 aludirk   20   0  412772 119868  73664 S   1.0   0.4   2:33.39 brave
 1825 aludirk   20   0 4671188 496536 138136 S   0.7   1.5   4:31.03 gnome-shell
 3791 aludirk   20   0 9129544 367780 133556 S   0.7   1.1   3:32.15 brave
50583 aludirk   20   0   15340   4308   3212 R   0.7   0.0   0:00.06 top
 2821 aludirk   20   0 4728576 191576  79100 S   0.3   0.6   0:27.78 brave
 3291 aludirk   20   0 4715944 180560  98272 S   0.3   0.5   0:16.97 brave
 3855 aludirk   20   0 9049056 286204 103524 S   0.3   0.9   0:22.83 brave
 3969 aludirk   20   0  866528  67260  54112 S   0.3   0.2   0:02.46 brave
26325 root      20   0  377620   6720   5628 S   0.3   0.0   0:14.96 openfortivpn
49384 aludirk   20   0 4727820 180504 100668 S   0.3   0.5   0:02.97 brave
50390 root      20   0       0      0      0 I   0.3   0.0   0:00.11 kworker/3:0-events
    1 root      20   0  167864  11892   8500 S   0.0   0.0   0:03.98 systemd
    2 root      20   0       0      0      0 S   0.0   0.0   0:00.02 kthreadd
    3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
    4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
    6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-kblockd
    9 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
   10 root      20   0       0      0      0 S   0.0   0.0   0:00.28 ksoftirqd/0
   11 root      20   0       0      0      0 I   0.0   0.0   0:08.18 rcu_sched
   12 root      rt   0       0      0      0 S   0.0   0.0   0:00.05 migration/0
   13 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_inject/0
```

Isolate a core from scheduler using ***kernel-boot-parameter***

By setting kernel-boot-parameter `isolcpus` (not linux command), we can isolate it from the OS scheduler. Here is the optimal setting of production machine in Yubo, which runs **RedHat**. Most important settings are highlighted. Please do not apply these settings in development machine, as these settings are the result of tunning in a long run, careless mistakes may lead to a linux-reinstallation.

```
>> cat /proc/cmdline
```

```
BOOT_IMAGE=/vmlinuz-3.10.0-862.el7.x86_64
root=/dev/mapper/rhel-root ro
crashkernel=auto
rd.lvm.lv=rhel/root
rd.lvm.lv=rhel/swap
rhgb
quiet
LANG=en_HK.UTF-8
idle=poll
processor.max_cstate=1
no_hz_full=1-8
isolcpus=1-8
intel_pstate=disable
nosoftlockup
mce=ignore_ce
```

D. TCP Connectivity

What are the differences between TCP and UDP?

- TCP is connection oriented, UDP is a connectionless, which supports unicast and multicast
- TCP transmits data as a stream, UDP transmits data as packets
- TCP does error checking and recovery, while UDP exposes to ***message loss***, ***duplication*** and ***corruption***
- TCP does guarantee data order as a stream, while UDP exposes to ***packet re-order***
- UDP is theoretically faster

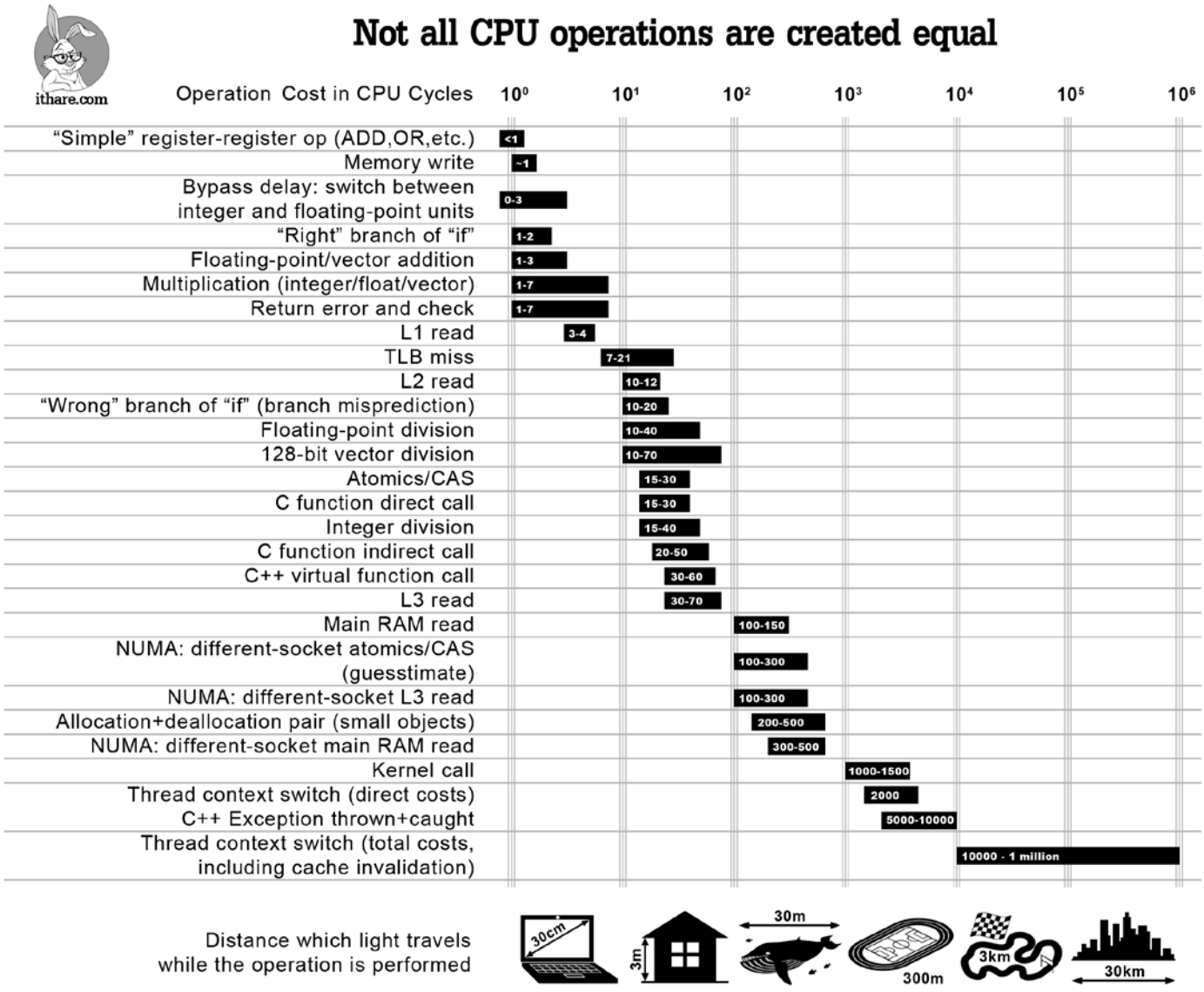
Can we implement TCP feature in UDP?

- implement error checking using checksum
- implement message order using sequence number

What is Nagle algorithm?

Nagle algorithm is an mechanism that :

- avoids traffic jam by too many small packets (with relatively large overhead-to-information ratio)
- it buffers and merges small packets together, sends them over when buffered messages is large enough
- improve bandwidth efficiency at the expense of latency
- can be disabled by ***No-delay option*** for low latency application, messages are sent immediately



For 1G Hz cpu, one clock cycle is 1.00 nano second. Light travels 0.3m
For 4G Hz cpu, one clock cycle is 0.25 nano second. Light travels 7.5cm