# CS3211 Tutorial 10

## Last Tutorial: Exam Revision
## T5 – Simon J

# Disclaimer: Exam questions in Quizzes

- Note that many past exam questions have become quiz questions, For examples, Q11 - Q14 in Quiz 1 are from Past Years Finals

- Therefore, we've covered a reasonable amount of them by now.

# What’s Important in CS3211

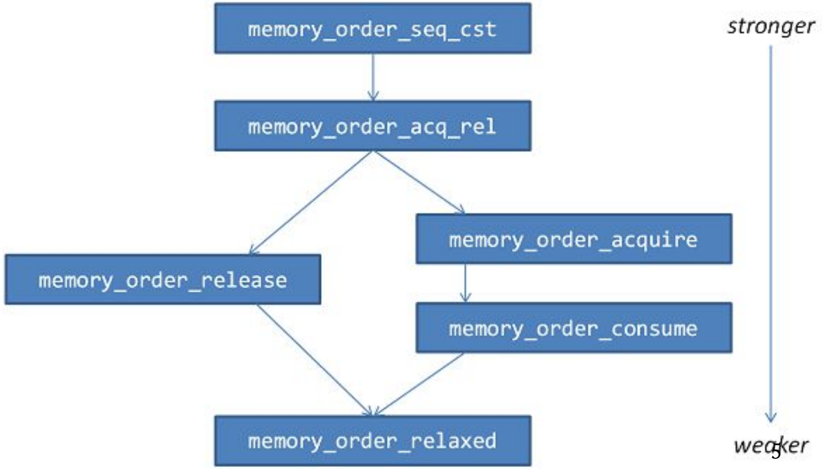
1. The most important thing in CS3211 Exam is **the idea** behind answering the question (i.e. the overview). Minor syntax details, minor calculation errors, etc. will not be penalized as long as your answer is coherent and clear.
    a. Therefore, minor syntaxes like #include, use::, etc. are not mandatory, they are just wasting your time.
2. **Argue clearly and smartly**. Use your time to explain your reasoning instead of copying the lecture codes.
    a. Reuse the ideas that are already mentioned in the Tut / Lectures. For example, Lock-Free Queue, Fan-In / Fan-Out Pattern, you can use the implementation in the lectures directly => auto LFQ = new LockFreeQueue();

# Memory model

- The modern C++ memory model exhibits a different behavior (to say, x86) when it runs on a weak-consistent architecture (e.g., ARM).

# Memory model

- The modern C++ memory model exhibits a different behavior when it runs on a weak-consistent *architecture* (e.g., ARM).
- False: the memory model exists to give us <u>guarantees</u> that we get same behavior on all architectures!



# Deadlocks without atomics?

- A program can get a deadlock while only using C++ atomics (assume the atomics are all implemented lock-free by the architecture)

# Deadlocks without atomics?

- A program can get a deadlock while only using C++ atomics (assume the atomics are all implemented lock-free by the architecture)
- True: just imagine a recursive spinlock acquisition

```
struct tas_lock {
  std::atomic<bool> lock_ = {false};
  void lock() { while(lock_.exchange(true, std::memory_order_acquire)); }
  void unlock() { lock_.store(false, std::memory_order_release); }
};
auto t = tas_lock{}
```

Thread 1: t.lock(); t.lock(); t.unlock(); // Recursive spinlock acquire – never gets second lock
Thread2: t.lock(); t.unlock(); // T1 and T2 both deadlocked

# Deadlocks without atomics?

- A program can get a deadlock while only using C++ atomics (assume the atomics are all implemented lock-free by the architecture)
- Actually, we also accepted false for those arguing via their own definition of deadlock
- Note: The definition of deadlock differs from textbook to textbook. Please specify the definition of deadlock that you are going to use if they ask you this in Exam, e.g.
  - Deadlock = Mutual Exclusion + No Progress + Hold/Wait + No Preemption
  - Deadlock = No Progress in the System / The System cannot transit from one state to another terminal state, etc.

# Unique vs Shared Ptr

- Image shows our lecture example of a threadsafe_queue
- Can we replace the unique_ptr by a shared_ptr without affecting correctness in this case?

```cpp
template<typename T>
class threadsafe_queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };
    std::mutex front_mutex;
    std::unique_ptr<node> front;
    std::mutex back_mutex;
    node* back;
    node* get_back()
    {
        std::lock_guard<std::mutex> back_lock(back_mutex);
        return back;
    }
    std::unique_ptr<node> pop_front()
    {
        std::lock_guard<std::mutex> front_lock(front_mutex);

        if(front.get()==get_back())
        {
            return nullptr;
        }
        std::unique_ptr<node> old_front=std::move(front);
        front=std::move(old_front->next);
        return old_front;
    }
public:
    threadsafe_queue():
        front(new node),back(front.get())
    {}
    threadsafe_queue(const threadsafe_queue& other)=delete;
    threadsafe_queue& operator=(const threadsafe_queue& other)=delete;
    std::shared_ptr<T> try_pop()
    {
        std::unique_ptr<node> old_front=pop_front();
        return old_front?old_front->data:std::shared_ptr<T>();
    }
    void push(T new_value)
    {
        std::shared_ptr<T> new_data(
            std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_back=p.get();
        std::lock_guard<std::mutex> back_lock(back_mutex);
        back->data=new_data;
        back->next=std::move(p);
        back=new_back;
    }
};
```

# Unique vs Shared Ptr

- Image shows our lecture example of a threadsafe_queue

- Can we replace the unique_ptr by a shared_ptr without affecting correctness in this case?
    - **Yes** – the shared_ptr behavior is a superset of a unique_ptr. It's just unnecessary overhead.
    - https://stackoverflow.com/questions/37884728/does-c11-unique-ptr-and-shared-ptr-able-to-convert-to-each-others-type

```cpp
template<typename T>
class threadsafe_queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };
    std::mutex front_mutex;
    std::unique_ptr<node> front;
    std::mutex back_mutex;
    node* back;
    node* get_back()
    {
        std::lock_guard<std::mutex> back_lock(back_mutex);
        return back;
    }
    std::unique_ptr<node> pop_front()
    {
        std::lock_guard<std::mutex> front_lock(front_mutex);

        if(front.get()==get_back())
        {
            return nullptr;
        }
        std::unique_ptr<node> old_front=std::move(front);
        front=std::move(old_front->next);
        return old_front;
    }
public:
    threadsafe_queue():
        front(new node),back(front.get())
    {}
    threadsafe_queue(const threadsafe_queue& other)=delete;
    threadsafe_queue& operator=(const threadsafe_queue& other)=delete;
    std::shared_ptr<T> try_pop()
    {
        std::unique_ptr<node> old_front=pop_front();
        return old_front?old_front->data:std::shared_ptr<T>();
    }
    void push(T new_value)
    {
        std::shared_ptr<T> new_data(
            std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_back=p.get();
        std::lock_guard<std::mutex> back_lock(back_mutex);
        back->data=new_data;
        back->next=std::move(p);
        back=new_back;
    }
};
```

# Quiz 1 / Finals Atomics Qns Review

# Atomics

| Thread 1 | Thread 2 |
|---|---|
| `x.store(1, memory_order_release);`<br>`y.store(2, memory_order_relaxed);` | `while (y.load(memory_order_relaxed)!=2);`<br>`cout << x.load(memory_order_acquire);` |

- Can the code result in an infinite loop?
- Can the code print 0?

# Atomics

| Thread 1 | Thread 2 |
|---|---|
| x.store(1, memory_order_release);<br>y.store(2, memory_order_relaxed); | while (y.load(memory_order_relaxed)!=2);<br>cout << x.load(memory_order_acquire); |

- Can the code result in an infinite loop? No! => C++11 Standard

11 Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.

- Can the code print 0? Yes!
  - `y.store` and `y.load` does not have synchronizes-with
  - Therefore, no happens-before for `x.store` and `x.load`

# Atomics

| Thread 1 | Thread 2 |
|---|---|
| `x.store(3, memory_order_relaxed);`<br>`z.store(4, memory_order_release);`<br>`x.store(5, memory_order_relaxed);` | `while (y.load(memory_order_acquire)!=2);`<br>`while (z.load(memory_order_acquire)!=4);`<br>`cout << x.load(memory_order_relaxed);` |

**Thread 3**

```
while (z.load(memory_order_acquire)!=4);
x.store(1, memory_order_relaxed);
y.store(2, memory_order_release);
```

- Can the code print 3?
- Can the code print 5?

# Atomics

| Thread 1 | Thread 2 |
|---|---|
| `x.store(3, memory_order_relaxed);`<br>`z.store(4, memory_order_release);`<br>`x.store(5, memory_order_relaxed);` | `while (y.load(memory_order_acquire)!=2);`<br>`while (z.load(memory_order_acquire)!=4);`<br>`cout << x.load(memory_order_relaxed);` |

| Thread 3 |
|---|
| `while (z.load(memory_order_acquire)!=4);`<br>`x.store(1, memory_order_relaxed);`<br>`y.store(2, memory_order_release);` |

- Can the code print 3? No
  - x.store(1) in Thread 3 happens-after x.store(3) in Thread 1
  - So 3 cannot ever be a valid value of x at the cout

# Atomics

| Thread 1 | Thread 2 |
| --- | --- |
| `x.store(3, memory_order_relaxed);`<br>`z.store(4, memory_order_release);`<br>`x.store(5, memory_order_relaxed);` | `while (y.load(memory_order_acquire)!=2);`<br>`while (z.load(memory_order_acquire)!=4);`<br>`cout << x.load(memory_order_relaxed);` |

| Thread 3 |
| --- |
| `while (z.load(memory_order_acquire)!=4);`<br>`x.store(1, memory_order_relaxed);`<br>`y.store(2, memory_order_release);` |

- **Can the code print 5? Yes**
  - x.store(5) is not part of the happens-before/after chain
  - It could be concurrently written to just before x.load - nothing restricts it

# 2023 data race qns: Where are the data races?

**For questions 14 – 18:** Consider 3 threads in C++ pseudo-code. There are 2 `std::atomic` variables, `x`, `y` and 2 non-atomic variables `z`, `t`. All variables were initialized to `0` before the threads were created. Assume the code will compile and run successfully.

| **Thread 1** | **Thread 2** |
|---|---|
| `z = 1;`<br>`t = 2;`<br>`y.store(3, memory_order_relaxed);`<br>`x.store(4, memory_order_release);`<br>`z = 5;` | `while`<br>`(x.load(memory_order_acquire)!=4);`<br>`cout << z;` |
| **Thread 3** | |
| `while (y.load(memory_order_relaxed)!=3);`<br>`cout << t;` | |

Are the following statements TRUE or FALSE? Optionally, you may justify your answers.

14. [1 mark] Thread 2 (z) will never print 0.
15. [1 mark] Thread 3 (t) will never print 0.
16. [1 mark] Thread 2 (z) might print 1.
17. [1 mark] Thread 3 (t) might print 2.
18. [1 mark] The code might run into an infinite loop.

# 2023 data race qns: Where are the data races?

**For questions 19 – 24:** Consider 3 threads in C++ pseudo-code. There are 2 `std::atomic` variables, `x`, `y` and a non-atomic variable `z`. All variables were initialized to `0` before the threads were created. Assume the code will compile and run successfully.

**Thread 1**

```
z = 2;
x.store(4, memory_order_release);
z = 3;
```

**Thread 2**

```
while
(y.load(memory_order_acquire)!=1);
cout << z;
```

**Thread 3**

```
if (x.load(memory_order_acquire)!=4) {
    y.store(1, memory_order_release); }
```

Are the following statements TRUE or FALSE? Optionally, you may justify your answers.

19. [1 mark] The code might print 0.
20. [1 mark] The code might print 2.
21. [1 mark] The code might print 3.
22. [1 mark] The code will never print 0.
23. [1 mark] The code might run into an infinite loop.
24. [1 mark] There is a data race.

# Quiz 2: Go Channels

Consider this Go program (assume it compiles correctly):

```
package main
import "fmt"

var c = make(chan int, 1)
var text string

func f() {
	text = "CS3211"
	<-c
}

func main() {
	go f()
	c <- 0
	fmt.Println(text)
}
```

What output will we see from this program?

Choose the corresponding option in case the code has some problems.

# Quiz 3

```
1  use std::thread;
2  use std::sync::{Mutex, Arc};
3
4  fn main() {
5    let counter = Arc::new(Mutex::new(0));
6
7    let t0 = {
8      let counter = counter.clone();
9      thread::spawn(move || {
10       // THE FOLLOWING IS NOT IDIOMATIC RUST
11       use std::ops::{Deref, DerefMut};
12
13       let mutex: &Mutex<i32> = counter.deref();
14       let lock_result = mutex.lock();
15       let mut lock_guard = lock_result.unwrap();
16       let counter_ref: &mut i32 = lock_guard.deref_mut();
17       *counter_ref += 1;
18       // END UNIDIOMATIC RUST
19     })
20   };
21   let t1 = {
22     let counter = counter.clone();
23     thread::spawn(move || {
24       *counter.lock().unwrap() += 1;
25     })
26   };
27
28   t0.join().unwrap();
29   t1.join().unwrap();
30
31   println!("{}", *counter.lock().unwrap());
32 }
```

After removing `move` from L9 and L23, can the new code compile? why or why not?

# IO_URING from 2023 PYP

# Q25: Identify Concurrency

[2 marks] Identify at least one part of the io_uring mechanism presented in Appendix 1 (page 14) that should handle concurrency safely.
For each part that you identified, explain why concurrent calls should be expected for that part.

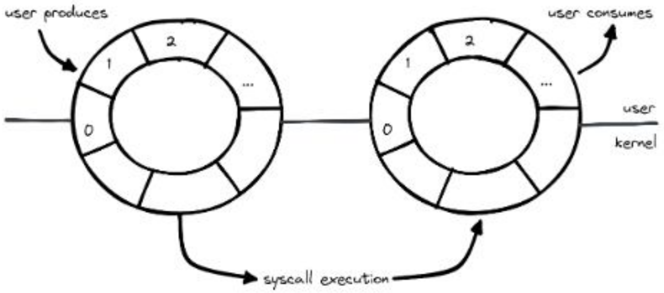
- Multiple threads need access to produce into the submission queue
- Kernel also needs to concurrently remove items from the SQ
- Vice-versa for the CQ

**Appendix 1: io_uring mechanism description for Part B.**

Io_uring is implemented as a kernel-level subsystem in Linux that provides an interface for performing asynchronous input/output (IO) operations. It is built on top of the existing asynchronous IO infrastructure and uses a ring buffer to manage IO requests and responses. (Note that this is a different mechanism from epoll from Lecture 10.)

The following are the key components of the io_uring subsystem:

1. Submission Queue (SQ): The submission queue is used to submit IO requests to the io_uring subsystem. It is a ring buffer that contains entries representing each IO request. An application can submit multiple IO requests in a single system call, which reduces the overhead of context switching between user and kernel space.
2. Completion Queue (CQ): The completion queue is used to retrieve completed IO requests from the io_uring subsystem. It is also a ring buffer that contains entries representing completed IO requests.
3. Polling: The io_uring subsystem uses the poll mechanism to wait for completed IO requests. When an IO request is completed, it is added to the completion queue, and the application can retrieve it using the poll system call.
4. Kernel Threads: The io_uring subsystem uses kernel threads to handle IO requests. When an application submits an IO request, it is added to the submission queue, and a kernel thread is responsible for processing it. This allows multiple IO requests to be processed concurrently, which improves performance.




Overall, the io_uring subsystem is implemented using a combination of kernel-level data structures, system calls, and threads to provide a scalable and efficient solution for performing asynchronous IO operations in Linux.

# Q26: Classic Problem

[2 marks] What classic synchronization problem can be used to solve the synchronization issues in io_uring mechanism presented in Appendix 1 (page 14)? Name the most similar problem and briefly explain why its solution can be used in this context.

**Producer–consumer**

# Q27: C++ Data Structure

Reuse of the  Lock Free Queue from tutorial, or implement fine grained locking for a double-ended queue.

**Write a C++ data structure** (class) that **implements the rings** (circular buffers) in the io_uring mechanism. Your structure, called **ConcurrentRing**, should:

- Safely support **concurrent submission** and **retrieval** of requests.
- Set a **maximum size** for the ring.
- Submission and retrieval requests should **block** once the number of pending requests in the ring has reached the maximum number.
- Your implementation may be optimized to allow for **high concurrency levels** (maximum 2 marks allocated for this requirement).

# Q28: C++ Server to use ConcurrentRing

Assume you are developing a high-performance server application in C++ that receives concurrent client connections and processes the clients' requests. **Write C++ code for this (following a template)**

- A client might send multiple requests during a session, as such it would be advisable to create a new thread for each client to handle the communication (read from and write to the client).
- Each client's request should be placed in a submit queue (SQ) implemented using a ConcurrentRing.
- The server should only have W workers that handle (process) the requests from the submit queue (SQ). Only these workers should call process() for each request.
- Once processed, the request result should be placed in a completion queue (CQ) implemented using a ConcurrentRing.
- Finally, each result from the completion queue should be sent back to the client that initiated the request.

```
int main() {
    // Set up TCP socket
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    // Bind socket to port
    listen(server_fd, 5);

    //Point D: Add your setup

    while (true) {
        // Accept new client connection
        int client_fd = accept(server_fd, nullptr, nullptr);

        //Point E: handle a client's request here;
        // It is not important what / how you read/write to the
        // client, use pseudocode such as "read(client_fd)", etc.
}
```

Correct (8)

28. Answer question 28 in the space provided.

```
int main() {
  int server_fd = socket(AF_INET, SOCK_STREAM, 0);
  listen(server_fd, 5);
  // Setup

  ConcurrentRing sq, cq;

  for (int i=0; i< W; i++){
    thread::spawn([&](){ // workers for processing request
      while(true){
        Request req = sq.retrieve_request();
        req.process(); // the result stored in request
        cq.submit_request(req);
      }
    };
    thread::spawn([&](){ // workers for sending back request
      while(true){
        auto req = cq.retrieve_request();
        auto client = req.client_fd;
        send(client, req);
      }
    };
  }

  while (true){
    int client_fd = accept(server_fd, nullptr, nullptr);
    thread::spawn([&](){
      auto data = read(client_fd);
      while(data){
        Request req;
        req.client_fd = client_fd;
        req.data = data;
        sq.submit_request(req);
        data = read(client_fd);
      }
    }
  }
}
```

General overview:

Have each client thread reads in request and puts into submission queu. W workers will be processing the requests and another W will send the done request back to client

Synchronization mechanism:

Not really needed since will be block if operation not allowed

Tasks concurrently:

Putting into sq, cq, Retrieving from sq, cq, reading, and sending can all be done concurrently

0177

# Q29: Go Version

- Here's SQ and CQ are channels

```
func main() {
     //Point F: add your own initialization, functions
     //calls, goroutines, etc.

     listener, err := net.Listen("tcp", ":8080")
     if err != nil {
        // handle error
     }

     for {
        conn, err := listener.Accept()
        if err != nil {
          // handle error
        }

    //Point G: add your own code to handle a connection
    //by calling Read and Write on conn. Conn is a
    // net.Conn type.
    }
}
```

9 / 10

Unnecessarily limits no. of workers (-1)



29. Answer question 29 in the space provided.

```
// Point F:
  sub_queue := make(chan Request, MAX_SIZE)
  com_queue := make(chan Request, MAX_SIZE)

  for i:=0 ; i < W; i++ {
      go func () {
        loop {
          select {
            case req := <- sub_queue
              req.process()
              com_queue <- req
          default:
          }
        }
      }
  }
  go func() {
    loop {
      select {
        case req := <-com_queue
            req.queue <- req
        default:
      }}
  }
// Point G:
  queue := make(chan Request)
  go func () {
    loop {
      req := conn.read ()
      req.queue = queue
      sub_queue <- req
  } }
  go func () {
    loop {
      select {
        case req := <- queue
        conn.send (req)
      } default:
  } }
```

Similar to Q28, we use buffered channel to achieve the same behavior as the Concurrent Ring. Each request has a channel associates to the client to send the request back. The parallelism in this implementation should also be the sam Here, we use loop/select pattern for reading from channels and pipelining which can be categorized into 3 stages: read, process and write.

ok

actually sends to right client



0162

# Q30: Rust mpsc version

- Here’s SQ and CQ are mpsc chans

```
use std::error::Error;
use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::mpsc;

enum IoOperation {
    Accept,
    Read(TcpStream, Vec<u8>),
    Write(TcpStream, Vec<u8>),
}
async fn process(data: &[u8]) -> Vec<u8> {
  // Process the data (in this example, we just echo it back)
  data.to_vec()
}


async fn main() -> Result<(), Box<dyn Error>> {
  let addr = SocketAddr::from(([127, 0, 0, 1], 8080));
  let listener = TcpListener::bind(&addr).await?;

  let (sq_tx, mut sq_rx) = mpsc::channel::<IoOperation>(100);
  let (cq_tx, mut cq_rx) = mpsc::channel::<IoOperation>(100);


  // Point H: add your own implementation

  // Example of how to accept a client’s connection
  // let (stream, _) = listener.accept().await
  // Use the TcpStream stream to communicate with that client

  // Example of how to read buf from a TcpStream stream
  // stream.read(&mut buf).await

  // Example of how to write buf to a TcpStream stream
  // stream.write_all(&buf).await


  Ok(())
}
```

ssing a little code
nalty (-1)

30. Answer question 30 in the space provided.

Point H:

```
while listener is active {
let sq_tx2 = sq_tx.clone()
  sq_tx2.send(Accept or Read or Write)
}

while !sq_rx.isempty {
    tokio::spawn(handle(sq_rx.recv))
    tokio::spawn(reply())
}

async handle(input: IOOperation){
  (switch input.type{= accept then listener.accept().await
                     = read then stream.read(&mut buf).await
  cq_tx2=cq_tx.clone()= write then stream.write_all(&buf).await
  cq_tx2.send(result)
}

async reply {
  loop (i = cq_rx.try_recv())
    i.stream.write_all
      (i.result).await
}
```

tries to send back to the right client

nice! Actually got the core of the qn

send IO requests to mpsc. Since mpsc can only have 1 reader, let one reader read the requests and send the tasks to be executed using tokio::spawn(handle). Each handle is accompanied by a tokio::spawn(reply) that receives from cq_rx and writes to the client.

number of concurrent tasks is limited by number of threads.

Hi Guys, that's all for the Tutorials :) Thank You for bearing with me for the past 4 months. Hope that we continue to keep in touch! ATB for your future endeavours 🔥

*Final Consultation*

1. *F2F: 23– 27 Apr*
2. *Online: Anytime*