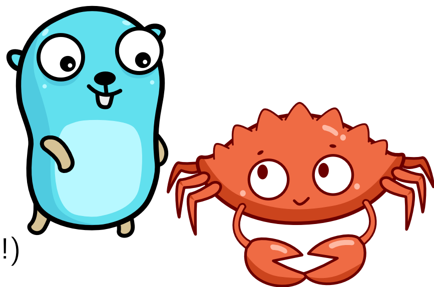
CS3211 Tutorial 10

Review, Exam Information

(AY 25/26 Semester 2)

April 16, 2026



(Compiled by Benson, thanks to all past and current TAs!)

Contents

Course Review

AY 22/23 Final Exam: io_uring

AY 24/25 Final Exam: Ticket Master

AY 24/25 Final Exam: Load Balancer

Contents

Course Review

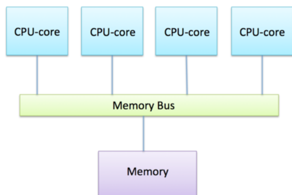
AY 22/23 Final Exam: io_uring

AY 24/25 Final Exam: Ticket Master

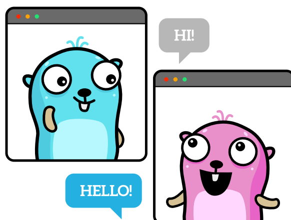
AY 24/25 Final Exam: Load Balancer

3 Different Paradigms

C++: Shared memory.



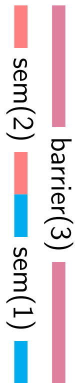
Go: Message passing.

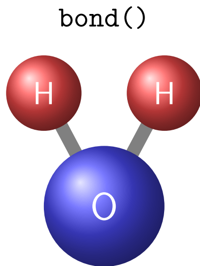


Rust: Async tasks.

C++: Using Synchronization Primitives

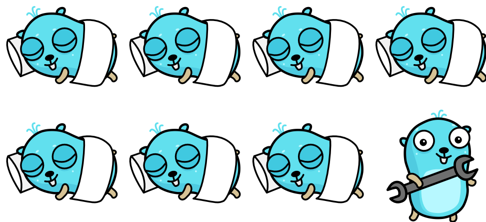
Semaphores + Barrier





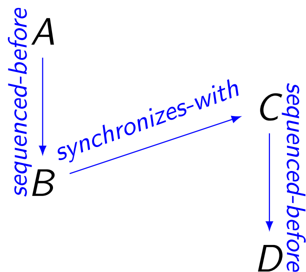
Monitors

```
1 while (/*condition not satisfied*/) {
2   cond.wait(lock);
3 }
```

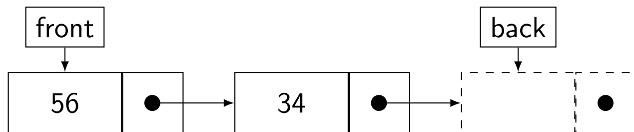


C++: Using Atomics / Lock-free Techniques

C++ Memory Model

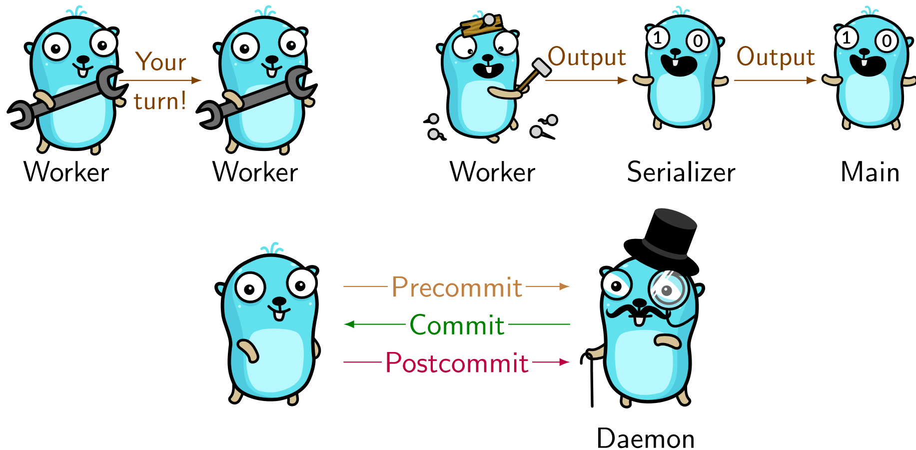


Lock Free Data Structures

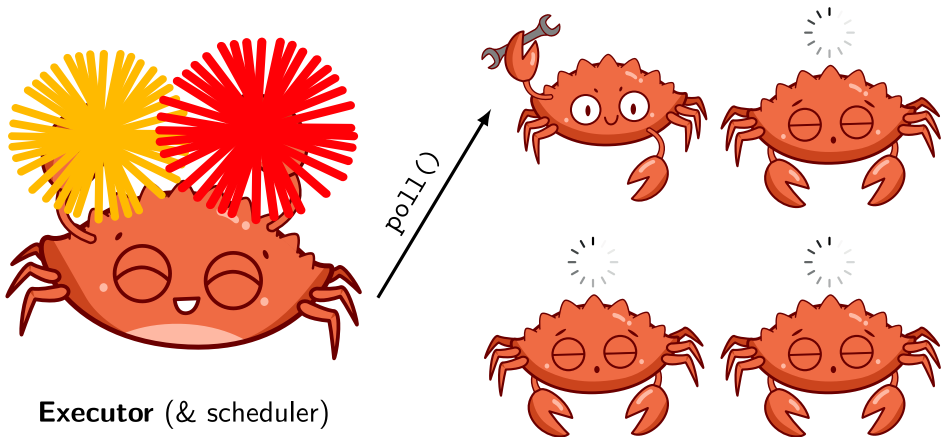



```
while (true) {
  old_value = value.load();
  // do something with old_value
  if (value.compare_exchange_weak(
          old_value, new_value)) {
    break;
  }
}
```

Go: Message Passing (Channels)



Rust: Asynchronous Programming



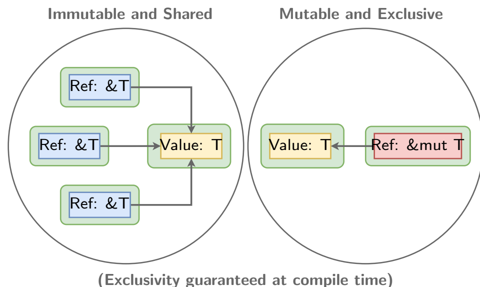
Programming languages matter!

Lifetime and RAII

```cpp
{
  std::scoped_lock lock{mut};
}
```

```rust
{
  let _permit = self.sem.acquire();
}
```

Safety



Random Things

- Most vexing parse in C++.

```cpp
Timer t1;
Timer t2(); // goes wrong!
Timer t3{};
```

- Loop variable scoping behaviour in Go.

```go
for i := 1; i <= 10; i++ {
  go func(){ fmt.Println(i); }()
}
```

Contents

Course Review

AY 22/23 Final Exam: io_uring

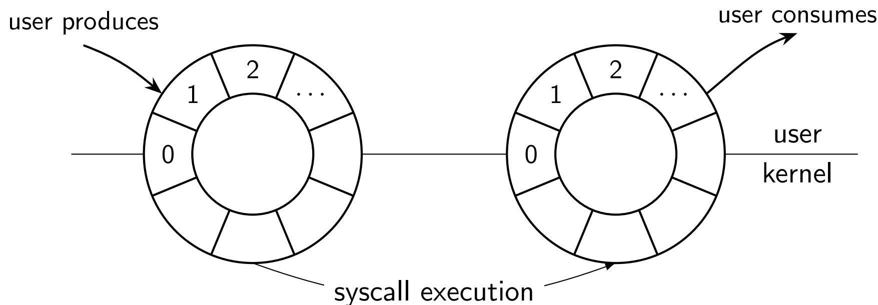
AY 24/25 Final Exam: Ticket Master

AY 24/25 Final Exam: Load Balancer

io_uring – Introduction

Io_uring is implemented as a kernel-level subsystem in Linux that provides an interface for performing asynchronous input/output (IO) operations.

- Polling: Completed IO requests are retried from the completion queue using the poll system call.
- Kernel Threads: Processes IO requests from the submission queue.



io_uring: Classical Synchronization Problem

Q26. 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

io_uring: Concurrent Ring

Q27. 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.

io_uring: Concurrent Ring

Pro-Tip 1

Follow the structure and points given (some can be empty).

```
struct Request {
// Point A: Add any data required for tracking client requests here
};

class ConcurrentRing {
private:
// Point B: Add any private variables here
public:
// Point C: Add any public variables here
void submit_request(Request req);
Request retrieve_request();
};
```

io_uring: Concurrent Ring

Pro-Tip 2

Using any data structures or algorithms implemented in class is fair game.

```
struct Request {
  int client_fd;
  data_t data;
  res_t result;
};

class ConcurrentRing {
private:
  LockFreeQueue<Request> queue;
  std::counting_semaphore write{SIZE};
  std::counting_semaphore read{0};
```

```
public:
  void submit_request(Request req) {
    write.acquire();
    queue.push(req);
    read.release();
  }
  Request retrieve_request() {
    read.acquire();
    std::optional<Request> req;
    while (true) {
      req = queue.try_pop();
      if (req) break;
    }
    write.release();
    return req.value();
  }
};
```

io_uring: Concurrent Ring

In your answer, explain and sketch C++ (pseudo-)code, including the following points:

- Give a general overview of your approach.
- What data structures and synchronization primitives will you use to implement your rings?
- Briefly describe the jobs (tasks) done concurrently. Be specific about what can potentially be executed in parallel (given the right hardware).
- What would be the maximum level of concurrency that you support in your implementation? (i.e. number of tasks that could potentially run in parallel)

io_uring: Concurrent Ring

Pro-Tip 3

Make sure you hit all the bullet points.

General Overview:

- Maintain a lock-free queue as implemented in Tutorial 4.
- Use two semaphores to enforce the maximum size of the ring.

Data structures and synchronization primitives:

- Data structure: Lock-free queue from Tutorial 4.
- Synchronization primitives: Two semaphores, write and read.

Tasks done concurrently:

- Multiple submission and retrieval can be done concurrently.

Maximum level of concurrency:

- The maximum size of the ring.

io_uring: Concurrent Ring

- [ ] N No implementation.
- [ ] N Question was misunderstood.
- [x] N Fine-grained implementation was attempted, but incorrect.
- C Correct
- C no answer/ fully incorrect
- [ ] N Level of concurrency is partially explained.
- [ ] N Fined-grained synchronization cannot be assessed as implementation is incomplete.

- [ ] N Synchronization is not fully explained, or partially incorrect.
- [x] N Level of concurrency is not explained or it is wrongly explained.
- [x] N Synchronization is broken (no prod-con)
- [ ] N No fine-grained synchronization
- [x] N The data structure is not behaving as a ring.

io_uring: C++ Server

Q28. Assume you are developing a high-performance server application in C++ that receives concurrent client connections and processes the clients' requests.

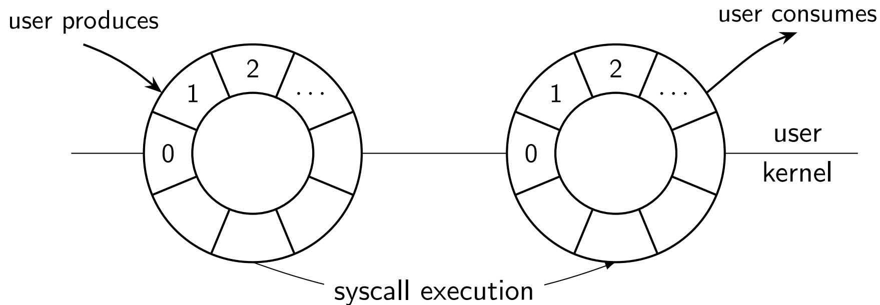
- Create a new thread for each client to handle the communication (read from and write to the client).
- Each client's request should be placed in a submit queue (SQ) implemented using a ConcurrentRing.
- The server should only have W workers that handle (process) the requests from the submit queue (SQ). 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.

io_uring: C++ Server

```
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.
  }
}
```

io_uring: C++ Server



io_uring: Concurrent Ring

Pro-Tip 4

Make sure the system is working as a whole (at least design-wise).

```
// Point D
ConcurrentRing SQ, CQ;

for (int i = 0; i < W; i++) {
  std::thread([&]() {
    while (true) {
      Request req = SQ.retrieve_req();
      req.process();
      CQ.submit_request(req);
    }
  });

  std::thread([&]() {
    while (true) {
      Request req = CQ.retrieve_req();
      send(req.client_fd, req);
    }
  });
}
```

```
while (true) {
  int client_fd = accept(...);

  // Point E
  std::thread([&]() {
    data_t data = read(client_fd);
    while (data) {
      Request req{client_fd, data};
      SQ.submit_request(req);
      data = read(client_fd);
    }
  });
}
```

io_uring: C++ Server

General Overview:

- Each client thread reads in requests and submit them to SQ.
- W worker threads processing the requests.
- Another W worker threads send the processed request back to client.

Synchronization mechanisms:

- Not needed if ConcurrentRing is implemented correctly.

Tasks done concurrently:

- Submission and retrieval of requests to/from SQ and CQ.
- Processing of requests.
- Sending of requests to clients.

io_uring: C++ Server

- [ ] N Minor incorrect order of operations
- [ ] N The explanation is correct, but no implementation is provided.
- [ ] N Unclear where the threads are started.
- C Correct
- [ ] N Workers process one request only.
- [ ] N Threads management is unclear/broken
- [x] N Multiple SQs or CQs
- [ ] N Client is not handled on a thread
- C No answer/ fully incorrect

- [x] N Requests processing is not implemented.
- [ ] N Each client is using 2 threads, one to read, one to write to the client
- [x] N Level of concurrency is not fully explained.
- [ ] N Incorrect sending of results to client
- [ ] N Level of concurrency not explained
- [ ] N Busy search of the result to send back to the client
- [ ] N Requests processing is not separated from client communication
- [ ] N Request result is not sent back to the client.

io_uring: Go Server

Q29. Sketch your implementation in Go.

- Submit queue (SQ) and Completion queue (CQ) are implemented using channels.

```go
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.
 }
}
```

io_uring: Go Server

```go
// Point F
SQ = make(chan Request, SIZE)
CQ = make(chan Request, SIZE)

for i := 0; i < W; i++ {
  go func() {
    for {
      select {
      case req := <-SQ:
        req.process()
        CQ <- req
      default:
      }
    }
  }()
}
```

```go
// Point F (cont.)
for i := 0; i < W; i++ {
  go func() {
    for {
      select {
      case req := <-CQ:
        req.queue <- req
      default:
      }
    }
  }()
}
```

```go
// Point G
queue := make(chan Request)

go func() {
  for {
    req := conn.read()
    req.queue = queue
    SQ <- req
  }
}()

go func() {
  for {
    select {
      case req := <-queue:
        conn.send(req);
      default:
    }
  }
}()
```

io_uring: Go Server

General Overview:

- Use buffered channel to achieve the same behavior as ConcurrentRing.
- Each client goroutine reads in requests and submit them to SQ.
- W worker goroutines to process and send back requests.
- To send back request correctly, each request is given a client channel.

Tasks done concurrently:

- Submission and retrieval of requests to/from SQ and CQ.
- Processing of requests.
- Maximum level of concurrency: Number of worker goroutines.

Go patterns:

- For-select: For sending and receiving request from channels.
- Pipelining: Three stages, read, process, and write.

io_uring: Go Server

- [x] N Responses not returned to correct client / not enough explanation
- [x] N Fan-in fan-out or pipeline pattern not mentioned
- [x] N Unnecessarily limits no. of workers
- [x] N No per-request concurrency
- [ ] N No per-client concurrency
- [ ] N No within-client concurrency / only handling 1 request per client / inefficient write back to CQ pattern / serial reads per-client-req but in right order
- [ ] N Doesn't use submit or completion queue (or multiple queues, or incorrect usage)

- [ ] N No / very limited code penalty but lots of other explanation
- [ ] N Severe lack of code and explanation

io_uring: Rust Server

Q30. Write a Rust implementation for the server.

- The server should place all the IO operations (accept connections, read client's requests, and write processed data back to client) in a submit queue (SQ).
- Submit queue (SQ) and Completion queue (CQ) are implemented using mpsc channel.

```
1  async fn main() -> Result<(), Box<dyn Error>> {
2    let addr = SocketAddr::from(([127, 0, 0, 1], 8080));
3    let listener = TcpListener::bind(&addr).await?;
4    let (sq_tx, mut sq_rx) = mpsc::channel::<IoOperation>(100);
5    let (cq_tx, mut cq_rx) = mpsc::channel::<IoOperation>(100);
6    // Point H: add your own implementation
7    Ok(())
8  }
```

io_uring: Rust Server

```rust
// Point H
loop {
  let (stream, _) =
      listener.accept().await?;

  tokio::spawn(async move {
    loop {
      let mut buf = Vec::new();
      stream.read(&mut buf).await;
      sq_tx.send(IoOperation::Read(
          stream, buf)).await;
    }
  });
}
```

```rust
tokio::spawn(async move {
  while let Some(op) = sq_rx.recv().await {
    tokio::spawn(async move {
      let res =
          process(op.data).await;
      cq_tx.send(IoOperation::Write(
          op.stream, res)).await;
    });
  }
});

tokio::spawn(async move {
  while let Some(op) = cq_rx.recv().await {
    op.stream.write_all(op.data).await;
  }
});
```

io_uring: Rust Server

General Overview:

- For each client, we spawn one tokio thread for handling incoming request and one tokio thread for sending back the result.
- Only one tokio thread can retrieve requests from SQ, since it is an mpsc (multi-producer-single-consumer) channel.
- Multiple tokio threads can handle processing of requests concurrently.

Programming paradigms:

- Pipelining: Three stages, read, process, and write.
- Fan-out/fan-in: Processing of read requests from SQ (fan-out) and submission of write requests to CQ (fan-in).

Tasks done concurrently:

- Processing of requests.

Maximum level of concurrency:

- The maximum size of the channels.

io_uring: Rust Server

- [ ] N Not handling multiple client requests
- [x] N Missing a little code penalty
- [ ] N Limited code penalty

- [ ] N Missing significant number of awaits (e.g., channel sends)

- [ ] N Not spawning enough extra tokio tasks for read/process/write / not using tokio::select intelligently

- [ ] N Missing one or two awaits
- [ ] N Writeback to wrong client / no clear attempt to do so
- [ ] N Incorrect use of sq/cq channels
- [ ] N No separate spawn for new processing tasks / limited number of tasks
- [ ] N No mention of async-await paradigm

Contents

Course Review

AY 22/23 Final Exam: io_uring

AY 24/25 Final Exam: Ticket Master

AY 24/25 Final Exam: Load Balancer

Ticket Master – Introduction

Ticketmaster's system is designed to handle high levels of concurrency during sale of tickets for events where thousands of users attempt to buy tickets simultaneously (for an event).

- User selects a seat from the live availability data.
- The system **temporarily reserves** the seat to prevent others from selecting it at the same time. This reservation is **time-bound**, allowing the user a short window to complete the purchase.
- If the user completes payment within the time limit, the seat is confirmed as **sold**; otherwise, it is **released** for others.

Ticket Master – C++ System

Q25. Complete the C++ class `TicketSystem`.

```cpp
// Point A
struct Seat {
  // Point B
};

class TicketSystem {
  vector<Seat> seats;
  // Point C

public:
  TicketSystem(int numSeats)
      : seats(numSeats) {
    // Point D
  }

  ~TicketSystem() {
    // Point E
  }
```

```cpp
  bool processPayment(int userID, int seatID) {
    // Point F
  }
  bool reserveSeat(int userID, int seatID,
                   int holdSeconds = 5) {
    steady_clock::time_point newExpiry =
        steady_clock::now() + seconds(holdSeconds);
    // Point G
  }
  bool confirmSeat(int seatID, int userID) {
    // Point H
  }
  void monitorExpirations() {
    while (true) {
      this_thread::sleep_for(seconds(1));
      auto now = steady_clock::now();
      // Point I
    }
  }
};
```

Ticket Master – C++ System

- Per-seat mutexes for concurrent safety.

```
struct Seat {
  // Point B
  mutex mtx; // For per-seat locking
  atomic<bool> sold = false;
  int reservedBy = -1;
  steady_clock::time_point expiryTime =
      steady_clock::time_point::min();
};
```

- Background thread to monitor expirations.

```
class TicketSystem {
  // Point C: expiration thread setup
  thread expirationThread;
  atomic<bool> running = true;

  TicketSystem(int numSeats)
      : seats(numSeats) {
    // Point D: Start monitor thread
    expirationThread =
        thread(&TicketSystem::
        monitorExpirations, this);
  }

  ~TicketSystem() {
    // Point E: Stop monitor thread
    running = false;
    if (expirationThread.joinable()) {
      expirationThread.join();
  }
}
```

Ticket Master – C++ System

```
bool processPayment(int userID, int seatID) {
  // Point F - empty
  confirmPayment(userID);
  return true;
}

bool reserveSeat(int userID, int seatID, int holdSeconds = 5) {
  // Point G - mutex + check time + check
  // (reservedBy + expiry for someone else holding seat)
  if (seatID < 0 || seatID >= seats.size()) return false;
  Seat& seat = seats[seatID];
  unique_lock<mutex> lock(seat.mtx);

  auto now = steady_clock::now();
  if (seat.sold || (seat.expiryTime > now &&
      seat.reservedBy != userID)) {
    return false; // Already sold or held by someone
  }

  seat.reservedBy = userID;
  seat.expiryTime = now + seconds(holdSeconds);
  return true;
}
```

Ticket Master – C++ System

```cpp
bool confirmSeat(int seatID, int userID) {
  // Point H
  if (seatID < 0 || seatID >= seats.size()) return false;
  Seat& seat = seats[seatID];
  unique_lock<mutex> lock(seat.mtx);
  auto now = steady_clock::now();
  if (seat.reservedBy == userID && seat.expiryTime > now) {
    seat.sold = true;
    seat.expiryTime = steady_clock::time_point::min();
    return true;
  } else {
    refundUser(userID);
    return false;
  }
}
```

Ticket Master – C++ System

```cpp
void monitorExpirations() {
  // Point I: Background expiration checker
  while (running) {
    this_thread::sleep_for(seconds(1));
    auto now = steady_clock::now();
    for (auto& seat : seats) {
      unique_lock<mutex> lock(seat.mtx);
      if (!seat.sold && seat.expiryTime < now) {
        seat.reservedBy = -1;
        seat.expiryTime =
            steady_clock::time_point::min();
      }
    }
  }
}
```

Ticket Master – C++ System

Briefly explain (through comments) why multiple users are allowed to select seats and pay at the same time (in parallel).

- Each Seat has its own mutex, so concurrent access to different seats is independent and thread-safe.
- Even if many users access the same seat, only one can lock and modify it at a time (mutex ensures this).

Ticket Master – C++ Atomics

Q26. Implement the start of `reserveSeat` using atomic variables.

```
struct Seat {
  atomic<bool> available;
  atomic<int> heldBy;
  Seat() : available(true), heldBy(0) {}
};
class TicketSystem {
  vector<Seat> seats;
  ...
  bool reserveSeat(int userID, int seatID,
                   int holdSeconds = 5) {
    steady_clock::time_point newExpiry =
        steady_clock::now() + seconds(holdSeconds);
    // Point J
  }
};
```

Ticket Master – C++ Atomics

We have to make sure **both** or **none** of the atomic variables are changed at the same time.

```
if (seatID < 0 || seatID >= seats.size()) return false;
// Attempt to atomically reserve the seat
bool expected = true;
if (seats[seatID].available.compare_exchange_strong(expected, false)) {
  // Successfully changed availability to false
  seats[seatID].heldBy.store(userID);
  return true;
} else {
  // Seat was not available
  return false;
}
```

Contents

Course Review

AY 22/23 Final Exam: io_uring

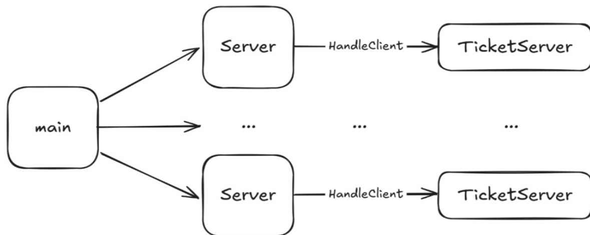
AY 24/25 Final Exam: Ticket Master

AY 24/25 Final Exam: Load Balancer

Load Balancer – Introduction

The LoadBalancer is responsible for distributing user (client) requests across multiple Servers.

- `LoadBalancer` dispatches the request to a `Server` based on the requested seat number.
- The details of reserving seats, handling payments, etc. have been provided in a type named `TicketServer`, and are handled by the function `HandleClient`.



Load Balancer – Go Implementation

```go
type Request struct {
  userID int
  seatID int
  responseCh chan bool
}

type Server struct {
  id int
  seatMin int
  seatMax int
  reqCh chan Request
  ts *TicketServer
}

type LoadBalancer struct {
  servers []*Server
}
```

```go
func NewServer(id, seatMin, seatMax int) *Server {
  s := &Server{
    id: id,
    seatMin: seatMin,
    seatMax: seatMax,
    reqCh: make(chan Request),
    ts: NewTicketServer(seatMax),
    // assume Server handles seatMin to seatMax
  }
  go func() {
    for req := range s.reqCh {
      success := s.ts.HandleClient(
          req.userID,req.seatID)
      req.responseCh <- success
    }
  }()
  return s
}
```

Load Balancer – Go Implementation

```go
type Request struct {
  userID int
  seatID int
  responseCh chan bool
}

type Server struct {
  id int
  seatMin int
  seatMax int
  reqCh chan Request
  ts *TicketServer
}

type LoadBalancer struct {
  servers []*Server
}
```

```go
func NewLoadBalancer(numServers int, maxSeatID int)
    *LoadBalancer {
  lb := &LoadBalancer{}
  seatsPerServer := (maxSeatID + numServers - 1) /
                   numServers
  for i := 0; i < numServers; i++ {
    seatMin := i * seatsPerServer
    seatMax := seatMin + seatsPerServer - 1
    if seatMax > maxSeatID {
      seatMax = maxSeatID
    }
    lb.servers = append(lb.servers,
        NewServer(i, seatMin, seatMax))
  }
  return lb
}
```

Load Balancer – Go Implementation

```go
type Request struct {
  userID int
  seatID int
  responseCh chan bool
}

type Server struct {
  id int
  seatMin int
  seatMax int
  reqCh chan Request
  ts *TicketServer
}

type LoadBalancer struct {
  servers []*Server
}
```

```go
func (lb *LoadBalancer) Dispatch(userID int,
    seatID int, responseCh chan bool) {
  // ---- Point E: Dispatch ----
  for _, s := range lb.servers {
    if seatID >= s.seatMin && seatID <= s.seatMax {
      s.reqCh <- Request{
        userID: userID,
        seatID: seatID,
        responseCh: responseCh,
      }
      return
    }
  }
}
```

Load Balancer – Go Implementation

```go
type Request struct {
  userID int
  seatID int
  responseCh chan bool
}

type Server struct {
  id int
  seatMin int
  seatMax int
  reqCh chan Request
  ts *TicketServer
}

type LoadBalancer struct {
  servers []*Server
}
```

```go
func main() {
  N := 3
  maxSeatID := 9

  // ---- Initialize LoadBalancer ----
  lb := NewLoadBalancer(N, maxSeatID)
  results := make([]chan bool, 20)
  for i := 1; i <= 20; i++ {
    seatID := rand.Intn(10)
    results[i-1] = make(chan bool)
    // ---- Point G: Dispatch ----
    lb.Dispatch(i, seatID, results[i-1])
  }
  for i := 1; i <= 20; i++ {
    // ---- Point H: Print results ----
    confirmed := <-results[i-1]
    fmt.Printf("User %d seat confirmation: %v\n",
        i, confirmed)
  }
  time.Sleep(5 * time.Second)
}
```

Load Balancer – Terminating Goroutines

Q28. How would you gracefully terminate the Server goroutines once all ticket sales for an event are completed, ensuring no request is left unprocessed? Explain how you would minimally update your implementation from the previous question to account for termination of goroutines.

`context.Context` + New select case in Server goroutines + `sync.WaitGroup`

Student Feedback Exercise:

Your Voice Matters!

Provide your feedback now >>

https://blue.nus.edu.sg/blue/

That's it!

All the best for the final exam!

Attendance

Session ID: 1940679
CS3211 Tutorial 10
16 April 2026 14:00-16:00
BIZ2-02-02 - SEMINAR ROOM 2-2