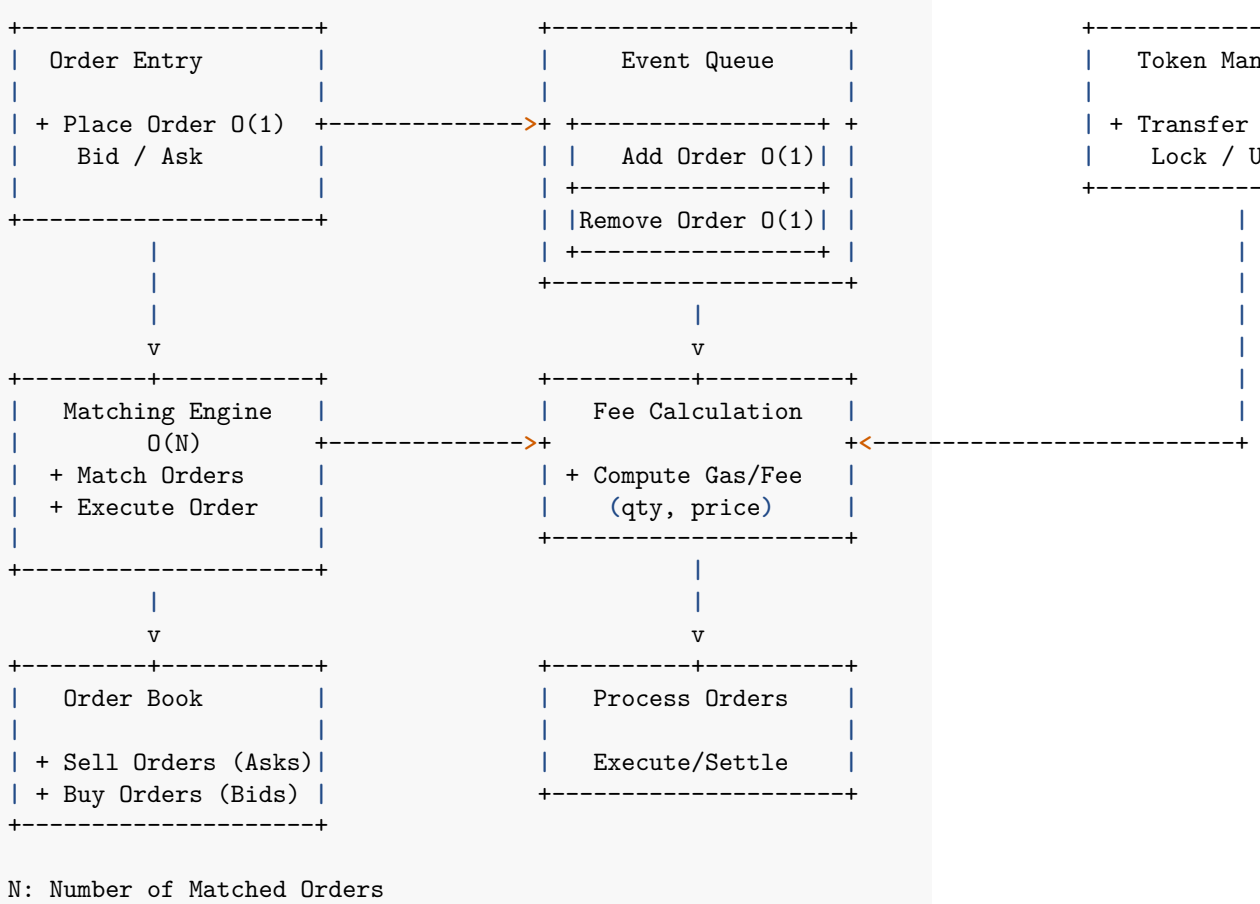
Chapter 3: Order Lifecycle on the OpenBook v1 Dex



Introduction

Welcome to the third chapter of this series where we explore the universe of web3 in Rust, particularly focusing on the Solana blockchain. In this chapter, we will explore the orders lifecycles on the OpenBook v1 Dex. If you have been following along, you may have noticed that I am working on a crate called **`openbook`**, which allows you to interact with any OpenBook v1 and v2 markets.

To start, let's explore event queues and how they process orders.

1. Event Queues

In the previous chapter, we have taken a look at the event queue which handles the asynchronous process of managing all the fills that occur when trades happen.

Whenever you place a new order on the book, you have to transfer some tokens to OpenBook, say WSOL or USDC, and whenever you make a trade, some tokens will transfer from locked state to freed state. Locked quantity and locked free quantity either **base** or **quote tokens** will **increment** and **decrement** based on the position that occurs.

```
pub struct OpenOrders {
    // ...snip...
    pub native_coin_free: u64, // base tokens
    pub native_coin_total: u64,

    pub native_pc_free: u64, // Quote tokens
    pub native_pc_total: u64,

    // ...snip...
}


impl OpenOrders {
    // ...snip...
    fn credit_locked_coin(&mut self, native_coin_amount: u64) {
        self.native_coin_total = self
            .native_coin_total
            .checked_add(native_coin_amount)
            .unwrap();
    }

    fn credit_locked_pc(&mut self, native_pc_amount: u64) {
        self.native_pc_total = self.native_pc_total.checked_add(native_pc_amount).unwrap();
    }

    fn lock_free_coin(&mut self, native_coin_amount: u64) {
        self.native_coin_free = self
            .native_coin_free
            .checked_sub(native_coin_amount)
            .unwrap();
    }

    fn lock_free_pc(&mut self, native_pc_amount: u64) {
        self.native_pc_free = self.native_pc_free.checked_sub(native_pc_amount).unwrap();
    }

    pub fn unlock_coin(&mut self, native_coin_amount: u64) {
        self.native_coin_free = self
            .native_coin_free
            .checked_add(native_coin_amount)
```

```
            .unwrap();
        assert!(self.native_coin_free <= self.native_coin_total);
    }

    pub fn unlock_pc(&mut self, native_pc_amount: u64) {
        self.native_pc_free = self.native_pc_free.checked_add(native_pc_amount).unwrap();
        assert!(self.native_pc_free <= self.native_pc_total);
    }
    // ...snip...
```

Reference: openbook-dex/program

For instance, if you place an order on the order book, your lock quantity will increase, and when you make a trade, your free quantity will increase, so you can withdraw these tokens.

```
    // ...snip...
    fn process_new_order_v3(args: account_parser::NewOrderV3Args) -> DexResult {
            // ...snip...

            let native_coin_unlocked = coin_unlocked.checked_mul(coin_lot_size).unwrap();
            let native_coin_credit = coin_credit.checked_mul(coin_lot_size).unwrap();
            let native_coin_debit = coin_debit.checked_mul(coin_lot_size).unwrap();

            open_orders_mut.credit_locked_coin(native_coin_credit);
            open_orders_mut.unlock_coin(native_coin_credit);
            open_orders_mut.unlock_coin(native_coin_unlocked);

            open_orders_mut.credit_locked_pc(native_pc_credit);
            open_orders_mut.unlock_pc(native_pc_credit);
            open_orders_mut.unlock_pc(native_pc_unlocked);

            open_orders_mut.native_coin_total = open_orders_mut
                .native_coin_total
                .checked_sub(native_coin_debit)
                .unwrap();
            open_orders_mut.native_pc_total = open_orders_mut
                .native_pc_total
                .checked_sub(native_pc_debit)
                .unwrap();

            // ...snip...
    }
    // ...snip...
```

Reference: openbook-dex/program

Event queues in a DEX play a crucial role in managing the lifecycle of orders. They ensure that all events such as order placements, trades, and cancellations are processed in a sequential and efficient manner. This sequential processing is important because it maintains the order integrity and ensures that no events are lost or processed out of order. When an order is placed, it is added to the event queue. This order stays in the queue until it is processed, which could mean matching it with an existing order, partially filling it, or canceling it.

```
impl<'ob> OrderBookState<'ob> {
    fn new_bid(
        &mut self,
        params: NewBidParams,
        event_q: &mut EventQueue,
        to_release: &mut RequestProceeds,
    ) -> DexResult<Option<OrderRemaining>> {
        // ...snip...
        let provide_out = Event::new(EventView::Out {
            // ...snip...
        });
        event_q
            .push_back(provide_out)
            .map_err(|_| DexErrorCode::EventQueueFull)?;

        // ...snip...
        let maker_fill = Event::new(EventView::Fill {
            // ...snip...
        });
        event_q
            .push_back(maker_fill)
            .map_err(|_| DexErrorCode::EventQueueFull)?;
    }
    // ...snip...
}
```

Reference: openbook-dex/program

```
pub struct Queue<'a, H: QueueHeader> {
    header: RefMut<'a, H>,
    buf: RefMut<'a, [H::Item]>,
}

impl<'a, H: QueueHeader> Queue<'a, H> {
    // ...snip...
    pub fn push_back(&mut self, value: H::Item) -> Result<(), H::Item> {
        if self.full() {
            return Err(value);
        }
```

```
        let slot = ((self.header.head() + self.header.count()) as usize) % self.buf.len();
        self.buf[slot] = value;

        let count = self.header.count();
        self.header.set_count(count + 1);

        self.header.incr_event_id();
        Ok(())
    }

    // ...snip...
    pub fn pop_front(&mut self) -> Result<H::Item, ()> {
        if self.empty() {
            return Err(());
        }
        let value = self.buf[self.header.head() as usize];

        let count = self.header.count();
        self.header.set_count(count - 1);

        let head = self.header.head();
        self.header.set_head((head + 1) % self.buf.len() as u64);

        Ok(value)
    }
    // ...snip...
}
```

Reference: openbook-dex/program

To visualize how the event queue operates, let's consider the following diagram:

```
+---------------------+
|     Event Queue     |
+---------------------+
| + OrderPlaced       |
| + TradeExecuted     |
| + OrderCancelled    |
+---------------------+
         |
         v
+---------------------+
|   Event Processor   |
+---------------------+
| Handle OrderPlaced  |
| Handle TradeExecuted|
```

```
|Handle OrderCancelled|
+---------------------+
         |
         v
+---------------------+
|    Order Matching   |
+---------------------+
|   Match Buy/Sell    |
+---------------------+
         |
         v
+---------------------+
|  Execute/Cancel     |
+---------------------+
```

2. Order Matching Engine

The order matching engine is the heart of any DEX. It matches buy and sell orders and ensures that trades are executed efficiently and fairly. In OpenBook, the order matching engine uses a priority queue to manage orders based on price-time priority as we have discussed above.

Order Matching

The matching engine processes orders by matching incoming buy and sell orders based on predefined rules. Buy orders are matched with sell orders of equal or lower price, and sell orders are matched with buy orders of equal or higher price. The engine ensures that the best available price is always given to both buyers and sellers, maximizing the efficiency and fairness of the market. This process involves checking the order book, which maintains a list of all outstanding **buy** and **sell** orders.

```rust
pub struct OrderBookState<'a> {
    pub bids: &'a mut Slab,
    pub asks: &'a mut Slab,
    pub market_state: &'a mut MarketState,
}
```

Reference: openbook-dex/program

To illustrate the process of the order matching engine, let's consider the following diagram:

```
+---------------------+
|     Order Book      |
+---------------------+
|  + Buy Orders       |
```

```
|  + Sell Orders      |
+---------------------+
         |
         v
+---------------------+
|  Matching Engine    |
+---------------------+
| Check Buy/Sell Match|
+---------------------+
         |
         v
+---------------------+
|   Execute Trade     |
+---------------------+
| Update Order Book   |
+---------------------+
```

3. Order Placement and Cancellation

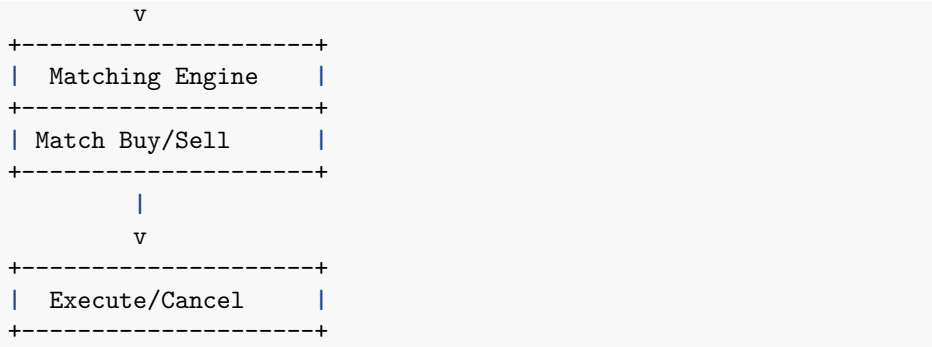
Order placement and cancellation are fundamental operations in any trading system. These actions allow traders to manage their positions and strategies effectively.

Placing an Order

Placing an order involves creating an order object with details such as quantity, price, and type (buy or sell). This order is then added to the order book, awaiting matching with a corresponding buy or sell order. Placing an order is the first step in interacting with the market and is crucial for liquidity and price discovery.

To better understand the order placement and cancellation process, let's consider the following diagram:

```
+---------------------+
|   Order Manager     |
+---------------------+
| Place Order         |
| Cancel Order        |
+---------------------+
         |
         v
+---------------------+
|    Order Book       |
+---------------------+
| Add Order / Remove  |
+---------------------+
         |
```



4. Order Execution and Settlement

Order execution and settlement are critical stages in the trading lifecycle. Execution refers to the process of completing a trade, while settlement involves the transfer of assets between parties (e.g. wallet and market vault).

Order Execution

Executing an order means fulfilling the trade as per the market conditions. When a buy order matches a sell order, the trade is executed, and the quantities and prices are updated accordingly. This ensures that both parties receive their respective assets as per the agreed terms of the trade.

To visualize the execution and settlement process, let's consider the following diagram:

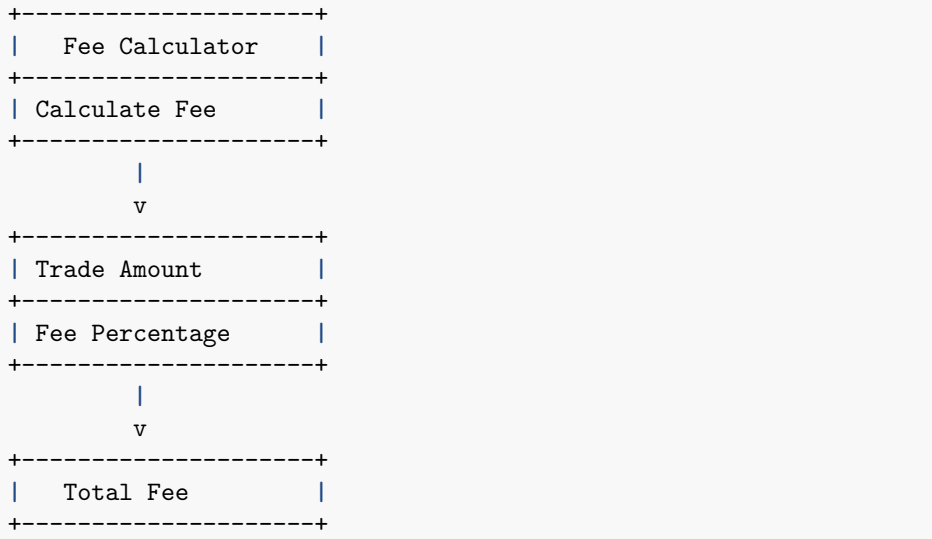
5. Fee Calculation and Distribution

Fees are an essential aspect of trading on a DEX. They incentivize liquidity providers and cover operational costs.

Fee Calculation Explained

Fee calculation involves determining the cost associated with each trade. This cost is a percentage of the trade amount. Calculating fees accurately is essential for transparency and fairness in trading.

To understand the fee calculation process, Let's consider the following diagram:



6. OpenBook Crate

TODO: The following code snippet utilizes the `openbook` crate for placing orders on OpenBook V1.

Conclusion

In this chapter, we explored various aspects of the order lifecycle on the OpenBook v1 Dex, including event queues, order matching, placement and cancellation, execution and settlement, and fee calculation. By understanding these components, you can build robust and efficient trading systems on the Solana blockchain in pure Rust.