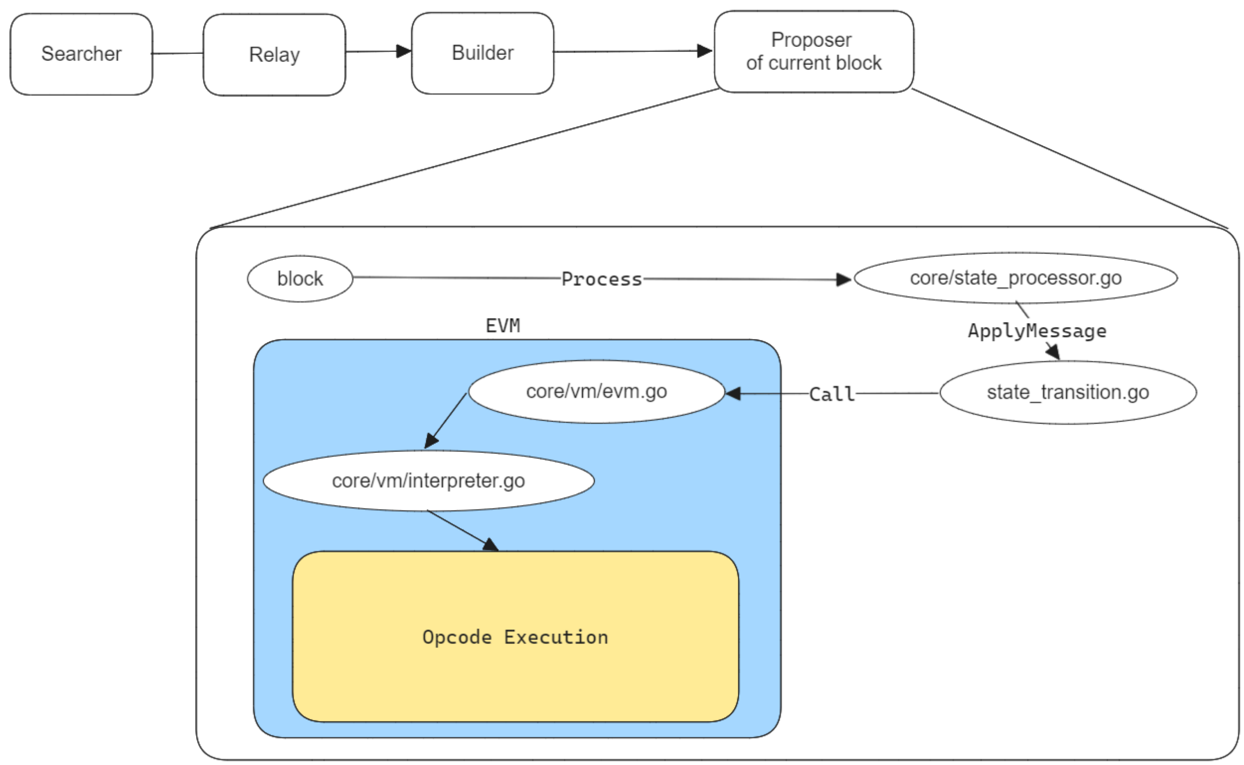
How a transaction is executed in Geth

High-level Flow (not the entire flow)



Code Analysis

Let’s track the **general process of transaction execution after a block is composed,** through geth codes.

1. **core/state_processor.go**
    - There is a function called `Process` , which processes the state changes of a given block (maybe built by block builder) and returns the receipts and logs.

```
func (p *StateProcessor) Process(block *types.Block, statedb *state.StateDE

   ...
```

- And transactions of the block are executed in the following loop within `Process` function.

```
for i, tx := range block.Transactions() {
      msg, err := TransactionToMessage(tx, signer, header.BaseFee)
      if err != nil {
         return nil, nil, 0, fmt.Errorf("could not apply tx %d [%v]: %w", i, tx.Has
      }
      statedb.SetTxContext(tx.Hash(), i)
      receipt, err := applyTransaction(msg, p.config, gp, statedb, blockNumbe
      if err != nil {
         return nil, nil, 0, fmt.Errorf("could not apply tx %d [%v]: %w", i, tx.Has
      }
      receipts = append(receipts, receipt)
      allLogs = append(allLogs, receipt.Logs...)
   }
```

- The main logic is in **`applyTransaction`**
- `TransactionToMessage` is in state_transition.go, which is a function to convert `Transaction` object to `Message` , making the client easier to deal with transaction information. Without `Message` , it will be much more difficult to process data since:
    - `Transaction` has a more complex structure than `Message` has.
    - `Transaction` is a form of interface and calling methods from it is slightly slower than calling a method directly on a `Message` struct.

▼ Implementation of `TransactionToMessage`

```
func TransactionToMessage(tx *types.Transaction, s types.Signer, ba
    msg := &Message{
        Nonce:            tx.Nonce(),
        GasLimit:         tx.Gas(),
        GasPrice:         new(big.Int).Set(tx.GasPrice()),
        GasFeeCap:        new(big.Int).Set(tx.GasFeeCap()),
        GasTipCap:        new(big.Int).Set(tx.GasTipCap()),
        To:               tx.To(),
        Value:            tx.Value(),
        Data:             tx.Data(),
        AccessList:       tx.AccessList(),
        SkipAccountChecks: false,
        BlobHashes:       tx.BlobHashes(),
        BlobGasFeeCap:    tx.BlobGasFeeCap(),
    }
    // If baseFee provided, set gasPrice to effectiveGasPrice.
    if baseFee != nil {
        msg.GasPrice = cmath.BigMin(msg.GasPrice.Add(msg.GasTipCa
    }
    var err error
    msg.From, err = types.Sender(s, tx)
    return msg, err
}
```

- `SetTxContext` function at statedb.go is called, storing current tx hash and index to StateDB.
- Each transaction is executed through `applyTransaction` .
    - It takes `vmenv` as one of its parameters, which is a snapshot of EVM in the current state (current block / current tx).

```
func applyTransaction(msg *Message, config *params.ChainConfig, gp *Gas
    // Create a new context to be used in the EVM environment.
    txContext := NewEVMTxContext(msg)
```

```
evm.Reset(txContext, statedb)

// Apply the transaction to the current state (included in the env).
result, err := ApplyMessage(evm, msg, gp)
if err != nil {
    return nil, err
}

// Update the state with pending changes.
var root []byte
if config.IsByzantium(blockNumber) {
    statedb.Finalise(true)
} else {
    root = statedb.IntermediateRoot(config.IsEIP158(blockNumber)).Bytes()
}
*usedGas += result.UsedGas

// Create a new receipt for the transaction, storing the intermediate root an
// by the tx.
receipt := &types.Receipt{Type: tx.Type(), PostState: root, CumulativeGasU
if result.Failed() {
    receipt.Status = types.ReceiptStatusFailed
} else {
    receipt.Status = types.ReceiptStatusSuccessful
}
receipt.TxHash = tx.Hash()
receipt.GasUsed = result.UsedGas

if tx.Type() == types.BlobTxType {
    receipt.BlobGasUsed = uint64(len(tx.BlobHashes()) * params.BlobTxBlol
    receipt.BlobGasPrice = evm.Context.BlobBaseFee
}

// If the transaction created a contract, store the creation address in the re
if msg.To == nil {
    receipt.ContractAddress = crypto.CreateAddress(evm.TxContext.Origin,
```

```
    }

    // Set the receipt logs and create the bloom filter.
    receipt.Logs = statedb.GetLogs(tx.Hash(), blockNumber.Uint64(), blockHa
    receipt.Bloom = types.CreateBloom(types.Receipts{receipt})
    receipt.BlockHash = blockHash
    receipt.BlockNumber = blockNumber
    receipt.TransactionIndex = uint(statedb.TxIndex())
    return receipt, err
}
```

- The main execution logic in on `ApplyMessage` , which is written on state_transition.go.

2. [core/state_transition.go]
    - The state transition is being calculated in the following logic:
        1. Nonce Increment
        2. Pre pay gas
        3. Create a new state object if the recipient is `nil`
        4. Value (ETH) transfer

           (If tx is creating a contract)

            1. Attempt to run tx data
            2. If valid, use result as code for the new state object
        5. Run Script section
        6. Derive new state root

    - The main operation of state transition are implemented in Step 5: Run Script section, with the function named `ApplyMessage` .
    - `ApplyMessage` calls `TransitionDB` function. It transits the state by applying the current message and returning the execution result (used gas, returndata,

execution error)

- The main state transition is executed by the below lines:

```
// Increment the nonce for the next transaction
st.state.SetNonce(msg.From, st.state.GetNonce(sender.Address())+1)
ret, st.gasRemaining, vmerr = st.evm.Call(sender, st.to(), msg.Data, st.gas
```

- This invokes `Call` function at core/vm/evm.go.

3. [`core/vm/evm.go`]

- The `Call` function executes the contract associated with the address with the given `msg.data`
- After some necessary checks, it brings the bytecode of the contract and calls `Run` function at core/vm/interpreter.go.

```
contract := NewContract(caller, AccountRef(addrCopy), value, gas)
contract.SetCallCode(&addrCopy, evm.StateDB.GetCodeHash(addrCopy)
ret, err = evm.interpreter.Run(contract, input, false)
```

4. [`core/vm/interpreter.go`]

- The `Run` function evaluates the contract's code with the given input data.
- Let's analyze the main for loop code line by line

```
for {
        if debug {
            // Capture pre-execution values for tracing.
            logged, pcCopy, gasCopy = false, pc, contract.Gas
        }
        // Get the operation from the jump table and validate the stack to ens
        // enough stack items available to perform the operation.
        op = contract.GetOp(pc)
```

```
operation := in.table[op]
cost = operation.constantGas // For tracing
// Validate stack
if sLen := stack.len(); sLen < operation.minStack {
    return nil, &ErrStackUnderflow{stackLen: sLen, required: operatio
} else if sLen > operation.maxStack {
    return nil, &ErrStackOverflow{stackLen: sLen, limit: operation.max
}
if !contract.UseGas(cost) {
    return nil, ErrOutOfGas
}
if operation.dynamicGas != nil {
    // All ops with a dynamic memory usage also has a dynamic gas c
    var memorySize uint64
    // calculate the new memory size and expand the memory to fit
    // the operation
    // Memory check needs to be done prior to evaluating the dynami
    // to detect calculation overflows
    if operation.memorySize != nil {
        memSize, overflow := operation.memorySize(stack)
        if overflow {
            return nil, ErrGasUintOverflow
        }
        // memory is expanded in words of 32 bytes. Gas
        // is also calculated in words.
        if memorySize, overflow = math.SafeMul(toWordSize(memSize)
            return nil, ErrGasUintOverflow
        }
    }
    // Consume the gas and return an error if not enough gas is availa
    // cost is explicitly set so that the capture state defer method can
    var dynamicCost uint64
    dynamicCost, err = operation.dynamicGas(in.evm, contract, stack
    cost += dynamicCost // for tracing
    if err != nil || !contract.UseGas(dynamicCost) {
        return nil, ErrOutOfGas
```

```
        }
        // Do tracing before memory expansion
        if debug {
            in.evm.Config.Tracer.CaptureState(pc, op, gasCopy, cost, callC
            logged = true
        }
        if memorySize > 0 {
            mem.Resize(memorySize)
        }
    } else if debug {
        in.evm.Config.Tracer.CaptureState(pc, op, gasCopy, cost, callCon
        logged = true
    }
    // execute the operation
    res, err = operation.execute(&pc, in, callContext)
    if err != nil {
        break
    }
    pc++
}
```

- This works as a while loop in other language
- Basically, the bytecode of a contract is composed up of various opcodes.

```
// Bytecode of 0xE592427A0AEce92De3Edee1F18E0157C05861564
// (Uniswap Router)
0x608060405260043610610112576000356 0e01c8...

// This is interpreted as...
0x / 60 / 80 / 60 / 40 / 52 / ...

// 60: PUSH1 - This takes 80 as its parameter
// ⇒ 6080 means PUSH 80 to the stack
```

```
// Next 6040 means PUSH 40 to the stack
// This is setting up free memory pointer
```

- So these lines are setting up opcodes, reading the bytecode of the contract.

  ```
  op = contract.GetOp(pc)
  operation := in.table[op]
  ```

  Table means Jumptable: It is a data structure or operations, and each subscript it contains corresponds to an EVM instruction.

  So above code is taking a corresponding instruction (opcode) from the contract's bytecode.

- After some necessary checks regarding stack validation, it executes the following code:

  ```
  res, err = operation.execute(&pc, in, callContext)
  ```

- Now that's almost there! We found the place where opcode is really executed.