Recap

World state: set of accounts identified by 32-byte address.

Two types of accounts:

(1) **owned accounts**: address = H(PK)

(2) **contracts**: address = H(CreatorAddr, CreatorNonce)
Recap: Transactions

- **To**: 32-byte address  (0 → create new account)
- **From**: 32-byte address
- **Value**: # Wei being sent with Tx
- **Tx fees** (EIP 1559): `gasLimit`, `maxFee`, `maxPriorityFee`
- **data**: what contract function to call & arguments
  
  - if To = 0: create new contract  `code = (init, body)`
- **[signature]**: if Tx initiated by an owned account
Recap: Blocks

Validators collect Tx from users:

⇒ run them sequentially on current world state

⇒ new block contains *updated world state*, Tx list, log msgs
The Ethereum blockchain: abstractly

prev hash → updated world state → accts. → prev hash

prev hash → Tx → log messages → prev hash

prev hash → updated world state → accts. → prev hash

prev hash → Tx → log messages → prev hash
Write code in Solidity (or another front-end language)

⇒ compile to EVM bytecode
   (other projects use WASM or BPF bytecode)

⇒ validators use the EVM to execute contract bytecode in response to a Tx
The EVM
The EVM

Stack machine (like Bitcoin) but with JUMP
- max stack depth = 1024
- program aborts if stack size exceeded; block proposer keeps gas
- contract can create or call another contract

In addition: two types of zero initialized memory
- **Persistent storage** (on blockchain): SLOAD, SSTORE (expensive)
- **Volatile memory** (for single Tx): MLOAD, MSTORE (cheap)
- LOG0(data): write data to log

see https://www.evm.codes
Every instruction costs gas, examples:

SSTORE addr (32 bytes), value (32 bytes)

• zero $\rightarrow$ non-zero: 20,000 gas
• non-zero $\rightarrow$ non-zero: 5,000 gas (for a cold slot)
• non-zero $\rightarrow$ zero: 15,000 gas refund (example)

CREATE: $32,000 + 200 \times \text{(code size)}$ gas;

CALL gas, addr, value, args

SELFDESTRUCT addr: kill current contract (5000 gas)

Refund is given for reducing size of blockchain state
Gas calculation

Why charge gas?

• Tx fees (gas) prevents submitting Tx that runs for many steps.
• During high load: block proposer chooses Tx from mempool that maximize its income.

Old EVM: (prior to EIP1559, live on 8/2021)

• Every Tx contains a gasPrice “bid” (gas → Wei conversion price)
• Producer chooses Tx with highest gasPrice \( \text{max } \sum (\text{gasPrice} \times \text{gasLimit}) \)

⇒ not an efficient auction mechanism (first price auction)
Gas prices spike during congestion

GasPrice in Gwei:

20 Gwei = 20 × 10^{-9} ETH

Average Tx fee in USD

popular project launch
Gas calculation: EIP1559

Every block has a “baseFee”: the minimum gasPrice for Tx in the block

\[
\text{baseFee} \text{ is computed from } \text{total gas} \text{ in earlier blocks:}
\]

- earlier blocks at gas limit (30M gas) \(\Rightarrow\) base fee goes up 12.5%
- earlier blocks empty \(\Rightarrow\) base fee decreases by 12.5%

If earlier blocks at “target size” (15M gas) \(\Rightarrow\) baseFee does not change
A transaction specifies three parameters:

- **gasLimit**: max total gas allowed for Tx
- **maxFee**: maximum allowed gas price
- **maxPriorityFee**: additional “tip” to be paid to block proposer

Computed **gasPrice** bid (in Wei = $10^{-18}$ ETH):

$$\text{gasPrice} \leftarrow \min(\text{maxFee}, \text{baseFee} + \text{maxPriorityFee})$$

Max Tx fee:  \(\text{gasLimit} \times \text{gasPrice}\)
Gas calculation

(1) if gasPrice < baseFee: abort
(2) If gasLimit × gasPrice > msg.sender.balance: abort
(3) deduct gasLimit × gasPrice from msg.sender.balance

(4) set Gas ← gasLimit
(5) execute Tx: deduct gas from Gas for each instruction
    if at end (Gas < 0): abort, Tx is invalid (proposer keeps gasLimit × gasPrice)

(6) Refund Gas × gasPrice to msg.sender.balance (leftover change)

(7) gasUsed ← gasLimit – Gas
    (7a) BURN gasUsed × baseFee
    (7b) Send gasUsed × (gasPrice – baseFee) to block producer
### Example baseFee and effect of burn

<table>
<thead>
<tr>
<th>block #</th>
<th>gasUsed</th>
<th>baseFee (Gwei)</th>
<th>ETH burned</th>
</tr>
</thead>
<tbody>
<tr>
<td>15763570</td>
<td>21,486,058</td>
<td>16.92</td>
<td>0.363</td>
</tr>
<tr>
<td>15763569</td>
<td>14,609,185</td>
<td>16.97</td>
<td>0.248</td>
</tr>
<tr>
<td>15763568</td>
<td>25,239,720</td>
<td>15.64</td>
<td>0.394</td>
</tr>
<tr>
<td>15763567</td>
<td>29,976,215</td>
<td>13.90</td>
<td>0.416</td>
</tr>
<tr>
<td>15763566</td>
<td>14,926,172</td>
<td>13.91</td>
<td>0.207</td>
</tr>
<tr>
<td>15763565</td>
<td>1,985,580</td>
<td>15.60</td>
<td>0.031</td>
</tr>
</tbody>
</table>

The table shows the relationship between gas used, base fee, and ETH burned in various blocks. The formula approximately equal to `gasUsed × baseFee` is highlighted.

- **baseFee < 16Gwei** ⇒ **new issuance > burn** ⇒ **ETH inflates**
- **baseFee > 16Gwei** ⇒ **new issuance < burn** ⇒ **ETH deflates**
Eth total supply (since merge)
Why burn ETH ???

EIP1559 goals (informal):
• users incentivized to bid their true utility for posting Tx,
• block proposer incentivized to not create fake Tx, and
• disincentivize off chain agreements.

Suppose no burn  (i.e., baseFee given to block producer):
⇒ in periods of low Tx volume proposer would try to increase volume by offering to refund the baseFee off chain to users.
Let’s look at the Ethereum blockchain etherscan.io:

<table>
<thead>
<tr>
<th>Latest Blocks</th>
<th>From/to address</th>
<th>Tx value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bk 15778674</td>
<td>Fee Recipient</td>
<td>0.088265 Ether</td>
</tr>
<tr>
<td>7 secs ago</td>
<td>Fee Recipient</td>
<td>0.088265 Ether</td>
</tr>
<tr>
<td>138 txns in 12 secs</td>
<td>0x52de8d3febd3a06d3c...</td>
<td>0.088265 Ether</td>
</tr>
<tr>
<td>Bk 15778673</td>
<td>Lido: Execution Layer Re...</td>
<td>0.2 Ether</td>
</tr>
<tr>
<td>19 secs ago</td>
<td>Lido: Execution Layer Re...</td>
<td>0.2 Ether</td>
</tr>
<tr>
<td>111 txns in 12 secs</td>
<td>0x404f5a67f72787a6dbd...</td>
<td>0.2 Ether</td>
</tr>
<tr>
<td>Bk 15778672</td>
<td>Flashbots: Builder</td>
<td>0 Ether</td>
</tr>
<tr>
<td>31 secs ago</td>
<td>Flashbots: Builder</td>
<td>0 Ether</td>
</tr>
<tr>
<td>313 txns in 12 secs</td>
<td>Optimism: State Commit...</td>
<td>0 Ether</td>
</tr>
<tr>
<td>Bk 15778671</td>
<td>Lido: Execution Layer Re...</td>
<td>0.14 Ether</td>
</tr>
<tr>
<td>43 secs ago</td>
<td>Lido: Execution Layer Re...</td>
<td>0.14 Ether</td>
</tr>
<tr>
<td>34 txns in 12 secs</td>
<td>Uniswap V3: Router 2</td>
<td>0.14 Ether</td>
</tr>
<tr>
<td>0x39feb77c9f90fae6196...</td>
<td>Uniswap V3: Router 2</td>
<td>0 Ether</td>
</tr>
<tr>
<td>0xb3336d324ed828dbc8...</td>
<td>Uniswap V3: Router 2</td>
<td>0 Ether</td>
</tr>
<tr>
<td>0xdeaf9880c1180b023...</td>
<td>Uniswap V3: Router 2</td>
<td>0 Ether</td>
</tr>
<tr>
<td>0x10c5a61426b506dcba...</td>
<td>Uniswap V2: Router 2</td>
<td>0 Ether</td>
</tr>
<tr>
<td>0xe1768151e16b86534a5d7...</td>
<td>Uniswap V2: Router 2</td>
<td>0 Ether</td>
</tr>
</tbody>
</table>
Let’s look at a transaction ...

Transaction ID: 0x14b1a03534ce3c460b022185b4 ...

From: 0x1deaf9880c1180b02307e940c1e8ef936e504b6a

To: Contract 0x68b3465833fb72a70ecdf485e0e4c7bd8665fc45 (Uniswap V3: Router 2)

Value: 0.14 Ether  ($182)

Data: Function: multicall() [calls multiple methods in a single call]

Contract generated a call to Contract 0xC02aaA39b22 ... (value:0.14)
Let’s look at the To contract ...

Contract 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
(Wrapped ETH: called from Uniswap V3: Router 2)

Balance: 4,133,236 Ether
Code: 81 lines of solidity

```solidity
function withdraw(uint wad) public {
    require(balanceOf[msg.sender] >= wad);
    balanceOf[msg.sender] -= wad;
    msg.sender.transfer(wad);
    Withdrawal(msg.sender, wad); // emit log event
}
```

anyone can read code snippet
Remember: contracts cannot keep secrets!

Contract 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
(Wrapped ETH)

Anyone can read contract state in storage array
⇒ never store secrets in contract!

Solidity variables stored in S[] array
Solidity

docs: https://solidity.readthedocs.io/

Several IDE’s available
interface IERC20 {
  function transfer(address _to, uint256 _value) external returns (bool);
  function totalSupply() external view returns (uint256);
  ...
}

contract ERC20 is IERC20 { // inheritance
  address owner;
  constructor() public { owner = msg.sender; }
  function transfer(address _to, uint256 _value) external returns (bool) {
    ...
    implementation ...
  }
}
Value types

- `uint256`
- `address (bytes32)`
  - `_address.balance`, `_address.send(value)`, `_address.transfer(value)`
  - call: send Tx to another contract
    ```solidity
    bool success = _address.call{value: msg.value/2, gas: 1000}(args);
    ```
  - delegatecall: load code from another contract into current context
- `bytes32`
- `bool`
Reference types

- structs
- arrays
- bytes
- strings
- mappings:
  - Declaration: mapping (address => unit256) balances;
  - Assignment: balances[addr] = value;

```
struct Person {
    uint128 age;
    uint128 balance;
    address addr;
}
Person[10] public people;
```
Globally available variables

- block: .blockhash, .coinbase, .gaslimit, .number, .timestamp
- gasLeft()
- msg: .data, .sender, .sig, .value
- tx: .gasprice, .origin
- abi: encode, encodePacked, encodeWithSelector, encodeWithSignature
- Keccak256(), sha256(), sha3()
- require, assert  e.g.: require(msg.value > 100, “insufficient funds sent”)

A ⇾ B ⇾ C ⇾ D:
at D:  msg.sender == C
tx.origin == A
Function visibilities

- external: function can only be called from outside contract.
  - Arguments read from calldata
- public: function can be called externally and internally.
  - if called externally: arguments copied from calldata to memory
- private: only visible inside contract
- internal: only visible in this contract and contracts deriving from it
- view: only read storage (no writes to storage)
- pure: does not touch storage

```plaintext
function f(uint a) private pure returns (uint b) { return a + 1; }
```
Inheritance

- **Inheritance**
  
  contract Destructable is owned {
    function destroy() public onlyOwner {
      selfdestruct(owner);
    }
  }

  code of contract “owned” is compiled into contract Destructable

- **Libraries**: library code is executed in the context of calling contract
  
  - library Search {
    function IndexOf();
  }
  
  - contract A {
    function B {
      Search.IndexOf();
    }
  }
ERC20 tokens


- A standard API for fungible tokens that provides basic functionality to transfer tokens or allow the tokens to be spent by a third party.

- An ERC20 token is itself a smart contract that maintains all user balances:
  ```solidity
mapping(address => uint256) internal balances;
```

- A standard interface allows other contracts to interact with every ERC20 token. No need for special logic for each token.
ERC20 token interface

- function `transfer(address _to, uint256 _value) external returns (bool);`
- function `transferFrom(address _from, address _to, uint256 _value) external returns (bool);`
- function `approve(address _spender, uint256 _value) external returns (bool);`
- function `totalSupply() external view returns (uint256);`
- function `balanceOf(address _owner) external view returns (uint256);`
- function `allowance(address _owner, address _spender) external view returns (uint256);`
How are ERC20 tokens transferred?

contract ERC20 is IERC20 {

    mapping (address => uint256) internal balances;

    function transfer(address _to, uint256 _value) external returns (bool) {
        require(balances[msg.sender] >= _value, "ERC20_INSUFFICIENT_BALANCE");
        require(balances[_to] + _value >= balances[_to], "UINT256_OVERFLOW");
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        emit Transfer(msg.sender, _to, _value);  // write log message
        return true;
    }

    Tokens can be minted by a special function mint(address _to, uint256 _value)
ABI encoding and decoding

- Every function has a 4 byte selector that is calculated as the first 4 bytes of the hash of the function signature.
  - In the case of `transfer`, this looks like `bytes4(keccak256("transfer(address,uint256")));`

- The function arguments are then ABI encoded into a single byte array and concatenated with the function selector. ABI encoding simple types means left padding each argument to 32 bytes.

- This data is then sent to the address of the contract, which is able to decode the arguments and execute the code.

- Functions can also be implemented within the fallback function
Calling other contracts

- Addresses can be cast to contract types.
  
  ```solidity
  address _token;
  IERC20Token tokenContract = IERC20Token(_token);
  ERC20Token tokenContract = ERC20Token(_token);
  ```

- When calling a function on an external contract, Solidity will automatically handle ABI encoding, copying to memory, and copying return values.
  
  ```solidity
  tokenContract.transfer(_to, _value);
  ```
Gas cost considerations

- Everything costs gas, including processes that are happening under the hood (ABI decoding, copying variables to memory, etc).

Considerations in reducing gas costs:
- How often do we expect a certain function to be called? Is the bottleneck the cost of deploying the contract or the cost of each individual function call?
- Are the variables being used in calldata, the stack, memory, or storage?
Stack variables

- Stack variables are generally the cheapest to use and can be used for any simple types (anything that is $\leq 32$ bytes).
  - `uint256 a = 123;`
- All simple types are represented as `bytes32` at the EVM level.
- Only 16 stack variables can exist within a single scope.
Calldata

- Calldata is a read-only byte array.

- Every byte of a transaction’s calldata costs gas
  
  (16 gas per non-zero byte, 4 gas per zero byte).
  
  ○ All else equal, a function with more arguments (and larger calldata) will cost more gas.

- It is cheaper to load variables directly from calldata, rather than copying them to memory.
  
  ○ For the most part, this can be accomplished by marking a function as `external`. 
Memory

- Memory is a byte array.
- Complex types (anything > 32 bytes such as structs, arrays, and strings) must be stored in memory or in storage.

```
string memory name = "Alice";
```

- Memory is cheap, but the cost of memory grows quadratically.
Storage

- Using storage is very expensive and should be used sparingly.
- Writing to storage is most expensive. Reading from storage is cheaper, but still relatively expensive.
- Mappings and state variables are always in storage.
- Some gas is refunded when storage is deleted or set to 0.
- Trick for saving has: variables < 32 bytes can be packed into 32 byte slots.
Event logs

- Event logs are a cheap way of storing data that does not need to be accessed by any contracts.
- Events are stored in transaction receipts, rather than in storage.
Security considerations

- Are we checking math calculations for overflows and underflows?
  - done by the compiler since Solidity 0.8.
- What assertions should be made about function inputs, return values, and contract state?
- Who is allowed to call each function?
- Are we making any assumptions about the functionality of external contracts that are being called?
Re-entrencency bugs
contract Bank{

    mapping(address=>uint) userBalances;

    function getUserBalance(address user) constant public returns(uint) {
        return userBalances[user];      }

    function addToBalance() public payable {
        userBalances[msg.sender] = userBalances[msg.sender] + msg.value;    }

    // user withdraws funds
    function withdrawBalance() public {
        uint amountToWithdraw = userBalances[msg.sender];

        // send funds to caller ... vulnerable!
        if (msg.sender.call{value:amountToWithdraw}() == false) { throw; }
        userBalances[msg.sender] = 0;
    }
}
contract Attacker {
    uint numIterations;
    Bank bank;

    function Attacker(address _bankAddress) { // constructor
        bank = Bank(_bankAddress);
        numIterations = 10;
        if (bank{value:75}.addToBalance() == false) { throw; } // Deposit 75 Wei
        if (bank.withdrawBalance() == false) { throw; } // Trigger attack
    }

    function () { // the fallback function
        if (numIterations > 0) {
            numIterations --; // make sure Tx does not run out of gas
            if (bank.withdrawBalance() == false) { throw; }
        }
    }
}
Why is this an attack?

(1) Attacker $\rightarrow$ Bank.addToBalance(75)

(2) Attacker $\rightarrow$ Bank.withdrawBalance $\rightarrow$
   Attacker.fallback $\rightarrow$ Bank.withdrawBalance $\rightarrow$
   Attacker.fallback $\rightarrow$ Bank.withdrawBalance $\rightarrow$ ...

withdraw 75 Wei at each recursive step
function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];

    userBalances[msg.sender] = 0;
    if (msg.sender.call{value:amountToWithdraw}() == false) {
        userBalances[msg.sender] = amountToWithdraw;
        throw;
    }
}
Next lecture: DeFi contracts