HỌC VIỆN KỸ THUẬT MẬT MÃ
KHOA ATTT

BÀI TẬP ĐÁNH GIÁ GIỮA KỲ
MÔN: CÔNG NGHỆ BLOCKCHAIN
Thời gian làm bài thi: 60 phút

NỘI DUNG

Bài 1. Trình bày quy trình viết và deploy smart contract lên mạng blockchain ETH

Quy trình viết và deploy smart contract lên mạng blockchain ETH là:

1. Môi trường phát triển:

Cài đặt một môi trường phát triển Ethereum như trình duyệt MetaMask hoặc Mist.

Cài đặt một trình biên dịch Solidity như Solidity Compiler hoặc Truffle Suite.

2. Viết smart contract:

Sử dụng một trình soạn thảo code để viết smart contract bằng ngôn ngữ Solidity, ngôn ngữ lập trình phổ biến cho việc phát triển smart contract trên Ethereum.

Viết mã của smart contract, xác định các hàm, biến và sự kiện cần thiết.

3. Biên dịch smart contract:

Sử dụng trình biên dịch Solidity để biên dịch mã nguồn Solidity thành mã bytecode có thể chạy trên Ethereum Virtual Machine (EVM).

4. Triển khai smart contract:

Mở môi trường phát triển Ethereum (như MetaMask) và kết nối với mạng Ethereum (ví dụ: mainnet hoặc testnet).

Tạo một ví Ethereum và có đủ số Ether để chi trả cho việc triển khai smart contract.

Sử dụng một công cụ triển khai smart contract như Remix hoặc Truffle để tải lên mã bytecode và triển khai smart contract lên mạng Ethereum.

5. Xác nhận triển khai:

Sau khi smart contract được triển khai, một giao dịch sẽ được gửi đến mạng Ethereum để xác nhận triển khai.

Giao dịch sẽ được khai thác bởi các thợ đào (miners) trên mạng Ethereum.

Giao dịch sẽ được xác nhận và kết quả triển khai smart contract sẽ được ghi vào blockchain Ethereum.

6. Giao tiếp với smart contract:

Sau khi smart contract được triển khai, bạn có thể giao tiếp với nó thông qua các giao dịch Ethereum.

Sử dụng địa chỉ smart contract để gọi các hàm và tương tác với dữ liệu của smart contract.

Bài 3. Viết một Smart Contract để quản lý hệ thống bỏ phiếu. Smart Contract này phải có các chức năng sau:

- Đăng ký ứng cử viên.
- Bỏ phiếu cho ứng cử viên.
- Xem kết quả bỏ phiếu

Bài làm

```
pragma solidity ^0.8.0;

contract VotingSystem {
    struct Candidate {
        string name;
        uint256 voteCount;
    }

    mapping(address => bool) private voters;
    Candidate[] private candidates;

    event CandidateRegistered(string name);
    event VoteCasted(address indexed voter, uint256
candidateIndex);
    event VotingResult(string winner, uint256 voteCount);

    function registerCandidate(string memory name) public {
        require(bytes(name).length > 0, "Invalid candidate name");
        candidates.push(Candidate(name, 0));
```

```
        emit CandidateRegistered(name);
    }

    function castVote(uint256 candidateIndex) public {
        require(candidateIndex < candidates.length, "Invalid candidate
index");
        require(!voters[msg.sender], "Already voted");
        candidates[candidateIndex].voteCount++;
        voters[msg.sender] = true;
        emit VoteCasted(msg.sender, candidateIndex);
    }

    function getVotingResult() public view returns (string memory,
uint256) {
        require(candidates.length > 0, "No candidates registered");
        uint256 maxVoteCount = 0;
        string memory winner;
        for (uint256 i = 0; i < candidates.length; i++) {
            if (candidates[i].voteCount > maxVoteCount) {
                maxVoteCount = candidates[i].voteCount;
                winner = candidates[i].name;
            }
        }
        return (winner, maxVoteCount);
    }

    function getCandidateCount() public view returns (uint256) {
        return candidates.length;
    }

    function getCandidate(uint256 index) public view returns (string
memory, uint256) {
        require(index < candidates.length, "Invalid candidate index");
        Candidate memory candidate = candidates[index];
        return (candidate.name, candidate.voteCount);
    }
}
```

Giải thích về smart contract trên:

Dòng 6: Khai báo struct Candidate để lưu trữ thông tin về ứng cử viên, bao gồm tên (name) và số phiếu (voteCount).

Dòng 8-9: Mapping voters được sử dụng để kiểm tra xem một địa chỉ đã bỏ phiếu hay chưa.

Dòng 11-12: Mảng candidates lưu trữ danh sách các ứng cử viên.

Dòng 14-16: Sự kiện CandidateRegistered được kích hoạt khi một ứng cử viên được đăng ký.

Dòng 17-19: Sự kiện VoteCasted được kích hoạt khi một bỏ phiếu được thực hiện.

Dòng 20-22: Sự kiện VotingResult được kích hoạt khi xem kết quả bỏ phiếu.

Dòng 24-28: Hàm registerCandidate() để đăng ký một ứng cử viên mới bằng cách cung cấp tên và lưu trữ thông tin về ứng cử viên trong mảng candidates. Sự kiện CandidateRegistered được kích hoạt.

Dòng 30-38: Hàm castVote() để bỏ phiếu cho một ứng cử viên bằng cách tăng số phiếu của ứng cử viên và đánh dấu người bỏ phiếu trong mapping voters. Sự kiện VoteCasted được kích hoạt.

Dòng 40-54: Hàm getVotingResult() trả về tên của ứng cử viên chiến thắng và số phiếu cao nhất. Hàm này duyệt qua danh sách ứng cử viên để tìm ra ứng cử viên có số phiếu cao nhất.

Dòng 56-60: Hàm getCandidateCount() trả về số lượng ứng cử viên đã đăng ký.

Dòng 62-68: Hàm getCandidate() trả về thông tin của ứng cử viên tại một vị trí cụ thể trong mảng `candidatesLưu ý rằng đây chỉ là một ví dụ cơ bản về smart contract quản lý hệ thống bỏ phiếu và có thể cần được cải tiến hoặc bổ sung tùy thuộc vào yêu cầu cụ thể của dự án. Bạn cần kiểm tra và thử nghiệm kỹ lưỡng trước khi triển khai smart contract trong môi trường blockchain thực tế.

Bài 4. Viết một Smart Contract để quản lý quỹ đầu tư. Smart Contract này phải có các chức năng sau:

- Đầu tư tiền vào quỹ.
- Rút tiền từ quỹ.
- Xem tổng số tiền quỹ hiện có.
- Phân phối lợi nhuận cho nhà đầu tư

Bài làm

```
pragma solidity ^0.8.0;
```

```
contract InvestmentFund {
    struct Investor {
        uint256 balance;
        uint256 lastProfitDistribution;
    }

    mapping(address => Investor) private investors;
    uint256 private totalBalance;
    uint256 private lastProfitDistribution;

    event Deposit(address indexed investor, uint256 amount);
    event Withdraw(address indexed investor, uint256 amount);
    event ProfitDistributed(uint256 amount);

    function deposit() public payable {
        require(msg.value > 0, "Invalid amount");
        investors[msg.sender].balance += msg.value;
        totalBalance += msg.value;
        emit Deposit(msg.sender, msg.value);
    }

    function withdraw(uint256 amount) public {
        require(amount > 0, "Invalid amount");
        require(amount <= investors[msg.sender].balance, "Insufficient
balance");
        investors[msg.sender].balance -= amount;
        totalBalance -= amount;
        payable(msg.sender).transfer(amount);
        emit Withdraw(msg.sender, amount);
    }

    function getTotalBalance() public view returns (uint256) {
        return totalBalance;
    }

    function distributeProfit(uint256 amount) public {
        require(amount > 0, "Invalid amount");
        require(amount <= address(this).balance, "Insufficient contract
balance");
```

```
        require(block.timestamp > lastProfitDistribution + 1 days,
"Cannot distribute profit yet");
        uint256 totalInvestedBalance = totalBalance;
        uint256 sharePerBalance = amount / totalInvestedBalance;
        for (address investorAddress : getAllInvestors()) {
            uint256 investorBalance =
investors[investorAddress].balance;
            uint256 profit = investorBalance * sharePerBalance;
            investors[investorAddress].balance += profit;
        }
        totalBalance += amount;
        lastProfitDistribution = block.timestamp;
        emit ProfitDistributed(amount);
    }

    function getInvestorBalance(address investorAddress) public
view returns (uint256) {
        return investors[investorAddress].balance;
    }

    function getLastProfitDistribution() public view returns (uint256)
{
        return lastProfitDistribution;
    }

    function getAllInvestors() public view returns (address[]
memory) {
        address[] memory investorAddresses = new
address[](totalInvestors());
        uint256 index = 0;
        for (address investorAddress : investors) {
            investorAddresses[index] = investorAddress;
            index++;
        }
        return investorAddresses;
    }

    function totalInvestors() public view returns (uint256) {
        return investors.length;
    }
```

```
}
```

Giải thích về smart contract trên:

- Dòng 6: Khai báo struct `Investor` để lưu trữ thông tin của nhà đầu tư, bao gồm số dư (`balance`) và thời điểm phân phối lợi nhuận cuối cùng (`lastProfitDistribution`).
- Dòng 8-9: Mapping `investors` được sử dụng để lưu trữ thông tin của các nhà đầu tư dựa trên địa chỉ của họ.
- Dòng 11-13: Biến `totalBalance` lưu trữ tổng số tiền trong quỹ.
- Dòng 14-15: Biến `lastProfitDistribution` lưu trữ thời điểm phân phối lợi nhuận cuối cùng.
- Dòng 17-19: Sự kiện `Deposit` được kích hoạt khi một nhà đầu tư gửi tiền vào quỹ.
- Dòng 20-22: Sự kiện `Withdraw` được kích hoạt khi một nhà đầu tư rút tiền từ quỹ.
- Dòng 23-25: Sự kiện `ProfitDistributed` được kích hoạt khi lợi nhuận được phân phối cho nhà đầu tư.
- Dòng 27-33: Hàm `deposit()` để nhà đầu tư gửi tiền vào quỹ bằng cách truyền vào giá trị Ether và cập nhật số dư của nhà đầu tư và tổng số tiền trong quỹ. Sự kiện `Deposit` đượckích hoạt.
- Dòng 35-47: Hàm `withdraw()` để nhà đầu tư rút tiền từ quỹ bằng cách truyền vào số tiền cần rút. Hàm này kiểm tra số dư của nhà đầu tư và chuyển tiền đến địa chỉ nhà đầu tư. Sự kiện `Withdraw` được kích hoạt.
- Dòng 49-53: Hàm `getTotalBalance()` trả về tổng số tiền trong quỹ.
- Dòng 55-78: Hàm `distributeProfit()` để phân phối lợi nhuận cho nhà đầu tư bằng cách truyền vào số tiền lợi nhuận cần phân phối. Hàm này tính toán lợi nhuận dựa trên tỷ lệ sở hữu của mỗi nhà đầu tư và cập nhật số dư của nhà đầu tư và tổng số tiền trong quỹ. Hàm này chỉ cho phép phân phối lợi nhuận sau mỗi khoảng thời gian (1 ngày trong ví dụ này). Sự kiện `ProfitDistributed` được kích hoạt.
- Dòng 80-84: Hàm `getInvestorBalance()` trả về số dư của một nhà đầu tư cụ thể.
- Dòng 86-90: Hàm `getLastProfitDistribution()` trả về thời điểm phân phối lợi nhuận cuối cùng.
- Dòng 92-107: Hàm `getAllInvestors()` trả về mảng chứa địa chỉ của tất cả nhà đầu tư.
- Dòng 109-113: Hàm `totalInvestors()` trả về số lượng nhà đầu tư.

Bài 6. Viết một Smart Contract để quản lý các chính sách bảo hiểm. Smart Contract này phải có các chức năng sau:

- Mua bảo hiểm.
- Yêu cầu bồi thường bảo hiểm.
- Phê duyệt hoặc từ chối yêu cầu bồi thường.
- Xem tổng số tiền bảo hiểm đã thu và đã trả.

Bài làm

```
pragma solidity ^0.8.0;

contract InsurancePolicy {
struct Policy {
uint256 premium;
uint256 coverage;
bool claimed;
bool approved;
}

mapping(address => Policy) private policies;
uint256 private totalPremiums;
uint256 private totalClaims;

event PolicyPurchased(address indexed policyHolder, uint256
premium, uint256 coverage);
event ClaimRequested(address indexed policyHolder, uint256
amount);
event ClaimApproved(address indexed policyHolder, uint256
amount);
event ClaimRejected(address indexed policyHolder);

function purchasePolicy() public payable {
require(policies[msg.sender].premium == 0, "Policy already
purchased");
uint256 premium = msg.value;
uint256 coverage = premium * 2;
Policy memory policy = Policy(premium, coverage, false, false);
policies[msg.sender] = policy;
totalPremiums += premium;
```

```
emit PolicyPurchased(msg.sender, premium, coverage);
}

function requestClaim() public {
require(policies[msg.sender].premium > 0, "No policy purchased");
require(!policies[msg.sender].claimed, "Claim already requested");
policies[msg.sender].claimed = true;
emit ClaimRequested(msg.sender, policies[msg.sender].coverage);
}

function approveClaim(address policyHolder) public {
require(policies[policyHolder].claimed, "No claim requested");
require(!policies[policyHolder].approved, "Claim already
approved");
policies[policyHolder].approved = true;
uint256 claimAmount = policies[policyHolder].coverage;
totalClaims += claimAmount;
payable(policyHolder).transfer(claimAmount);
emit ClaimApproved(policyHolder, claimAmount);
}

function rejectClaim(address policyHolder) public {
require(policies[policyHolder].claimed, "No claim requested");
require(!policies[policyHolder].approved, "Claim already
approved");
policies[policyHolder].claimed = false;
emit ClaimRejected(policyHolder);
}

function getTotalPremiums() public view returns (uint256) {
return totalPremiums;
}

function getTotalClaims() public view returns (uint256) {
return totalClaims;
}

function getPolicy(address policyHolder) public view returns
(uint256, uint256, bool, bool) {
Policy memory policy = policies[policyHolder];
```

```
        return (policy.premium, policy.coverage, policy.claimed,
        policy.approved);
        }
}
```

Giải thích về smart contract trên:

- Dòng 6: Khai báo struct `Policy` để lưu trữ thông tin về chính sách bảo hiểm, bao gồm mức phí (premium), mức bảo hiểm (coverage), trạng thái đã yêu cầu bồi thường (claimed) và đã được phê duyệt (approved).
- Dòng 8-11: Mapping `policies` được sử dụng để ánh xạ từ địa chỉ người mua chính sách tới thông tin chính sách bảo hiểm của họ.
- Dòng 12-13: Các biến `totalPremiums` và `totalClaims` để theo dõi tổng số tiền phí bảo hiểm đã thu và tổng số tiền bồi thường đã trả.
- Dòng 15-18: Sự kiện được kích hoạt khi một chính sách bảo hiểm được mua (`PolicyPurchased`), khi yêu cầu bồi thường được gửi (`ClaimRequested`), khi yêu cầu bồi thường được phê duyệt (`ClaimApproved`) và khi yêu cầu bồi thường bị từ chối (`ClaimRejected`).
- Dòng 20-28: Hàm `purchasePolicy()` để người dùng mua chính sách bảo hiểm bằng cách gửi Ether và lưu trữ thông tin chính sách bảo hiểm của họ trong mapping `policies`. Tổngsố phí bảo hiểm đã thu được tăng lên và sự kiện `PolicyPurchased` được kích hoạt.
- Dòng 30-36: Hàm `requestClaim()` để người dùng yêu cầu bồi thường bảo hiểm. Trạng thái `claimed` của chính sách bảo hiểm được đánh dấu là true và sự kiện `ClaimRequested` được kích hoạt.
- Dòng 38-48: Hàm `approveClaim()` để phê duyệt yêu cầu bồi thường bảo hiểm. Trạng thái `approved` của chính sách bảo hiểm được đánh dấu là true, tổng số tiền bồi thường tăng lên và tiền bồi thường được chuyển đến người mua chính sách. Sự kiện `ClaimApproved` được kích hoạt.
- Dòng 50-55: Hàm `rejectClaim()` để từ chối yêu cầu bồi thường bảo hiểm. Trạng thái `claimed` của chính sách bảo hiểm được đánh dấu là false và sự kiện `ClaimRejected` được kích hoạt.
- Dòng 57-61: Hàm `getTotalPremiums()` trả về tổng số tiền phí bảo hiểm đã thu.
- Dòng 63-67: Hàm `getTotalClaims()` trả về tổng số tiền bồi thường đã trả.

- Dòng 69-74: Hàm `getPolicy()` trả về thông tin chi tiết về chính sách bảo hiểm của một người dùng cụ thể, bao gồm mức phí, mức bảo hiểm, trạng thái đã yêu cầu bồi thường và đã được phê duyệt.

Bài 7. Viết một Smart Contract để quản lý ví tiền điện tử. Smart Contract này phải có các chức năng sau:

- Gửi tiền vào ví.
- Rút tiền từ ví.
- Xem số dư của ví

Bài làm

```
pragma solidity ^0.8.0;

contract ElectronicWallet {
    mapping(address => uint256) private balances;

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

    function withdraw(uint256 amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
        payable(msg.sender).transfer(amount);
    }

    function getBalance() public view returns (uint256) {
        return balances[msg.sender];
    }
}
```

Giải thích về smart contract trên:

Dòng 3: Sử dụng pragma để chỉ định phiên bản Solidity.

Dòng 5: Mapping balances để theo dõi số dư của từng địa chỉ.

Dòng 7-10: Hàm deposit() cho phép gửi tiền vào ví. Người gửi phải gửi một số Ether khi gọi hàm này, số Ether này được thêm vào số dư của địa chỉ người gửi.

Dòng 12-17: Hàm withdraw() cho phép rút tiền từ ví. Người dùng cần chỉ định số tiền cần rút và hàm sẽ kiểm tra xem số dư của địa chỉ người dùng có đủ để

rút hay không. Nếu đủ, số tiền sẽ được chuyển đến địa chỉ người dùng và số dư của địa chỉ đó sẽ giảm đi số tiền đã rút.

Dòng 19-22: Hàm getBalance() trả về số dư của địa chỉ người gọi hàm.