5. Transaction Execution 과정을 pseudocode로 나타내고, intrinsic gas와 upfront cost에 대해 자세히 서술해주세요.

1) Transaction Validity

이더리움 노드는 트랜잭션을 전송받으면 트랜잭션을 실행(execute)하기 전에 우선 해당 트랜잭션이 유효한지 `validity`를 먼저 검사한다. 트랜잭션 유효성 검사 과정은 다음과 같다.

1. 트랜잭션이 `RLP` 형식에 맞게 잘 인코딩되었는지 검사한다.
2. 트랜잭션의 서명`(signature)`을 검증한다.
3. 트랜잭션 `nonce`가 해당 `account`의 현재 `nonce`와 일치하는지 검사한다.
4. 트랜잭션의 `intrinsic gas`가 트랜잭션에 설정된 `gas limit` 보다 작은지 검사한다.
5. 트랜잭션 전송자의 `account balance`가 해당 트랜잭션에 요구되는 `upfront cost` 보다 크거나 같은지 검사한다.

> ***참고사항*** *:*
> *트랜잭션을 실행하기 전에 트랜잭션의 유효성 검사와는 별개로 검사해야 하는 한 가지 규칙이 더 있다. 전체 트랜잭션들에 사용되는 모든 gas를 합쳤을 때, 블록의 `gas limit`을 초과하지 않도록 해야 한다. 만약 블록의 `gas limit` 초과할 경우 해당 트랜잭션은 유효하더라도 블록에 넣을 수 없다.*

(1) RLP 형식 검사

`Recursive Length Prefix(RLP)`는 이더리움에서 사용하는 트랜잭션이나 스마트 컨트랙트와 같은 nested된 arrays 데이터를 `serialization`하기 위한 인코딩 방법이다.

따라서 트랜잭션을 처리하기 전에 우선 해당 트랜잭션이 `RLP` 규칙에 맞게 정확하게 인코딩되었는지 검사해야 한다.

(2) 서명 검증

해당 `account`에서 전송한 트랜잭션이 실제 `private key`를 가진 주인이 전송한 것인지 검사해야 하기 때문에 함께 전달된 전자서명을 검증해야 한다.

이더리움의 일반 `account address`는 사용자의 `public key`를 해쉬하여 생성된다. 전송된 서명으로부터 `public key`를 복원(recover)하여 해당 `account`와 비교하여 검사하고, 전제 트랜잭션 내용과 서명된 값이 일치하는지 검증하여 데이터가 위변조되지 않았는지 검증한다.

> ***참고사항*** *:*
> *이더리움 에서는 비트코인 과 마찬가지로 `Secp256k1` 타원곡선을 사용하는 `ECDSA` 알고리즘을 서명에 사용한다.*

(3) Account nonce 검사

이더리움은 비트코인과 다르게 `UTXO` 모델 이 아니라 `Account` 모델 을 사용하기 때문에 각 트랜잭션을 구분할 수 있는 별도의 방법이 필요하다. 그렇지 않으면 동일한 트랜잭션을 여러번 전송하여 `replay attack`이 가능하기 때문에 이전에 전송했던 트랜잭션과 구분할 수 있도록 `nonce`를 함께 전송해야 한다. 이 `nonce` 값은 지금까지 해당 `account`에서 전송된 트랜잭션 개수를 나타내며, 새로운 트랜잭션을 생성할 때 마다 `nonce` 값을 1씩 증가시켜 전송한다.

따라서 트랜잭션 안의 `nonce` 값이 해당 `account` 의 현재 `nonce` 값과 동일한지 검사해야 한다.

(4) Intrinsic gas 검사

이더리움 트랜잭션을 처리할 때 사용되는 gas에는 두 가지 종류가 있다. 하나는 `execution gas` 이고, 두 번째는 `intrinsic gas` 이다.

`execution gas` 는 해당 트랜잭션을 처리하기 위해 `EVM` 에서 수행되는 연산(operations)량에 의해 결정된다. 따라서 더 많은 연산이 사용될 수록 더 많은 gas가 소모된다.

`intrinsic gas` 는 트랜잭션에서 전송되는 `message call` 을 위한 `data` 또는 스마트 컨트랙트 생성을 위한 `EVM-code` 에 의해 계산되는 `gas` 이다.

만약 `Nzeros` 를 해당 데이터에 포함된 `zero bytes` 의 개수라고 하고, `Nnonzeros` 를 해당 데이터에 포함된 `non-zero bytes` 의 개수라고 한다다면, `intrinsic gas` 는 다음과 같이 계산된다.

```
intrinsic gas = (Gtxdatazero * Nzeros)
              + (Gtxdatanonzero * Nnonzeros)
              + Gtxcreate
              + Gtransaction
```

이더리움 Yellow Paper의 `Appendix G` 에는 다음과 같은 `Fee Schedule` 정보가 나와있다.

- Gtxdatazero = 4 gas
- Gtxdatanonzero = 68 gas
- Gtxcreate = 32000 gas
- Gtransaction = 21000 gas

`Fee Schedule` 에서 알 수 있듯이 `Gtxdatazero` 값이 `Gtxdatanonzero` 보다 작기 때문에 `data` 에 `zero bytes` 가 많을 수록 `intrinsic gas` 가 낮아지는 것을 알 수 있다.

결과적으로 이더리움 노드는 트랜잭션을 실행하기 전에 해당 트랜잭션에서 지불해야 하는 `intrinsic gas` 를 계산하여 해당 트랙잭션을 처리할 때 최대로 사용할 수 있는 `gas limit` 을 초과하지 않는지 검사해야 한다.

(5) Upfront cost 검사

사용자는 트랜잭션에 `gasLimit` 과 `gasPrice` 를 입력하여 전송한다. 이 때 `gasLimit` 은 해당 트랜잭션을 처리할 때 최대로 사용할 수 있는 gas의 양이고, `gasPrice` 는 사용되는 가스당 가격을 의미한다.

어떤 `account` 에서 트랜잭션을 전송했을 때 해당 트랜잭션을 실행하면서 소모될 수 있는 최대 비용은 `gasLimit * gasPrice` 비용과 기본적으로 전송되는 `value` 비용을 합친 값이 된다. 따라서 이 둘을 합친 최대 비용을 `upfront cost` 라고 부르며, `upfront cost` 는 다음과 같이 계산된다.

```
upfront cost = (gasLimit * gasPrice) + value
```

따라서 트랜잭션을 실행할 때 최대로 소모될 수 있는 `upfront cost` 계산하여 해당 `account` 에서 실제로 해당 비용을 지불할 수 있는 충분한 `account balance` 가 남아 있는지 검사해야 한다.

2) Transaction Execution

트랜잭션의 유효성 검사가 완료되면 트랜잭션을 실행해야 한다. 트랜잭션의 실행과정은 다음과 같다.

1. 트랜잭션을 전송한 `account` 의 `nonce` 를 1 증가시킨다.
2. 해당 `acocunt` 의 `balance` 를 `upfront cost` 만큼 감소시킨다.
3. 트랜잭션 실행을 위해 사용가능한 가스를 계산한다(`Gavailable` = `gasLimit` - `intrinsic gas`).
4. 트랜잭션의 연산(operations)을 실행한다(`value transfer` , `message call` 또는 `contract creation`).
5. `SELFDESTRUCT` 또는 `SSTORE` 연산에 대한 비용을 전송자에게 환불(refund)한다.
6. 모든 연산 후에 남은 gas를 사용자에게 환불한다.
7. 블록 마이닝에 대한 수수료를 해당하는 `beneficiary account` 에게 지불한다.

(1) Account nonce 증가

앞서 말한 것처럼 사용자는 트랜잭션을 생성할 때마다 `nonce` 가 1씩 증가한다. 따라서 해당 `account nonce` 를 1 증가시킨다.

(2) Upfront cost 차감

트랜잭션 실행 초기에 사용자의 `account balance` 에서 최대 사용가능한 `upfront cost` 차감한다. 그리고 나중에 모든 연산이 끝난 후에 남은 비용을 환불해준다.

(3) 사용가능한 gas 계산

트랜잭션을 실행하기 사용할 수 있는 gas를 계산한다. 최대 사용가능한 `gasLimit` 에서 `intrinsic gas` 을 차감하여 트랜잭션 실행에 사용할 수 있는 gas를 계산한다.

(4) 트랜잭션 실행(Transaction Execution)

기본적인 `value transfer` 또는 스마트 컨트랙트를 위한 `EVM operations` 를 실행한다. `EVM operation` 마다 소모되는 gas가 정해져 있으며, 이 값에 따라 각 `operation` 을 실행하며 소모되는 gas를 이전에 계산했던 사용가능한 gas에서 차감한다. 이 때 만약 사용가능한 gas가 부족할 경우 트랜잭션 실행에 실패한다.

(5) `SELFDESTRUCT` , `SSTORE` 연산에 대한 gas 환불

스마트 컨트랙트가 `SELFDESTRUCT` 연산을 실행하여 컨트랙트를 삭제하였을 경우 `storage` 공간 반환에 대한 보상으로 24000 gas를 호출자에게 환불해준다.

또한 만약 `SSTORE` 연산으로 `storage` 공간에 0를 입력할 경우에도 마찬가지로 15000 gas를 환불해준다.

(6) 모든 연산 후에 남은 gas 환불

모든 트랜잭션 실행 후에 남은 gas에 대해서는 다시 전송자에게 환불해준다.

(7) 마이닝 수수료 지불

마이너는 해당 트랜잭션을 실행하면서 소모되었던 모든 비용을 블록 마이닝에 대한 보상으로 지불받는다.

3) Transaction Execution Pseudocode

지금까지 설명한 `transaction validity` 과정과 `transaction execution` 과정을 JavaScript `pseudocode`로 나타내면 다음과 같다.

```
// tx-execution.js

const handleTransaction = (trasaction) => {
  const beneficiary;

  let tx; // transaction object
  let intrinsicGas;
  let upfrontCost;
  let gasAvailable;
  let gasRemained;
  let gasRefunded;
  let refundCost;
  let fee;


  // 1. Transaction Validity

  // check RLP encoding.
  try {
   tx = rlpDecode(trasaction);  // parse the transaction.
  } catch (e) {
    throw 'RLP decoding failed!';
  }

  // verify tx signature.
  if (!verifySignature(tx.sig, tx.account))
    throw 'Signature verification failed!';

  // check tx nonce.
  if (getAccountNonce(tx.account) !== tx.nonce)
    throw 'Transaction nonce is invalid!';

  // check intrinsic gas.
  intrinsicGas = calIntrinsicGas(tx);
  if (intrinsicGas >= tx.gasLimit)
    throw 'Gas limit is less than intrinsic gas!';

  // check upfront cost.
  upfrontCost = tx.value + gasLimit * gasPrice;
  if (getAccountBalance(tx.account) < upfrontCost)
    throw 'Account balance is less than upfront cost!';


  // 2. Transaction Execution
```

```
  // increase account nonce by 1.
  increaseAccountNonce(tx.account);

  // Subtract upfront cost from account balance .
  decreaseAccountBalance(tx.account, upfrontCost);

  // Calculate available gas.
  gasAvailable = tx.gasLimit - intrinsicGas;

  // Execute operations.
  try {
    gasRemained = executeTransaction(tx, gasAvailable);
  } catch (e) {
    // out of gas, revert, exceptions.
    throw 'Transaction execution failed!'
  }

  // Refund gas for 'SELFDESTRUCT`, 'SSTORE' operations.
  gasRefunded = calGasRefund(tx);

  // Refund the remaining gas.
  gasRefunded += gasRemained;

  // Calculate refund cost.
  refundCost = gasRefunded * tx.gasPrice;

  // Refund cost.
  increaseAccountBalance(tx.account, refundCost);

  // Pay mining fee.
  fee = upfrontCost - refundCost;
  increaseAccountBalance(beneficiary, fee);

  return true;
}
```

References

[1] *Vaibhav Saini. (2019, Nov 29).* *Transaction Execution — Ethereum Yellow Paper Walkthrough (4/7)* *[Hacker Noon]*

[2] *Dr. Gavin Wood. (2020, Sep 5).* *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER PETERSBURG VERSION* *[Ethereum]*