// SPDX-License-Identifier: MIT

// coder by 教父, tg:@bytedance888

pragma solidity ^0.6.2;

import "./DividendPayingToken.sol";

import "./SafeMath.sol";

import "./IterableMapping.sol";

import "./Ownable.sol";

import "./IUniswapV2Pair.sol";

import "./IUniswapV2Factory.sol";

import "./IUniswapV2Router.sol";

contract OKKAKA is ERC20, Ownable {

    using SafeMath for uint256;

    IUniswapV2Router02 public uniswapV2Router;

    address public  uniswapV2Pair;

    bool private swapping;

    ETHBackDividendTracker public dividendTracker;

    address public deadWallet = 0x000000000000000000000000000000000000dEaD;

    address public immutable ETH = address(0x5fAc926Bf1e638944BB16fb5B787B5bA4BC85b0A); //Reward ETH

    uint256 public swapTokensAtAmount = 10000000000 \* (10\*\*18);

    mapping(address => bool) public \_isbclisted;

    uint256 public ETHRewardsFee = 8;

    uint256 public liquidityFee = 1;

    uint256 public marketingFee = 3;

    uint256 public totalFees = ETHRewardsFee.add(liquidityFee).add(marketingFee);

    uint256 public \_maxTxAmount = 1000000000000000 \* (10\*\*18);

    uint256 public \_maxWalletAmount = 1000000000000000 \* (10\*\*18);

    bool public isL;

    uint256 public killNum = 0;

    uint256 public lunachB;

    address public \_marketingWalletAddress = 0xc45CF12d724ED17238e93da728A35E654DCEFa12;

    // use by default 300,000 gas to process auto-claiming dividends

    uint256 public gasForProcessing = 300000;

     // exlcude from fees and max transaction amount

    mapping (address => bool) private \_isExcludedFromFees;

    // store addresses that a automatic market maker pairs. Any transfer \*to\* these addresses

    // could be subject to a maximum transfer amount

    mapping (address => bool) public automatedMarketMakerPairs;

    event UpdateDividendTracker(address indexed newAddress, address indexed oldAddress);

    event UpdateUniswapV2Router(address indexed newAddress, address indexed oldAddress);

    event ExcludeFromFees(address indexed account, bool isExcluded);

    event ExcludeMultipleAccountsFromFees(address[] accounts, bool isExcluded);

    event SetAutomatedMarketMakerPair(address indexed pair, bool indexed value);

    event LiquidityWalletUpdated(address indexed newLiquidityWallet, address indexed oldLiquidityWallet);

    event GasForProcessingUpdated(uint256 indexed newValue, uint256 indexed oldValue);

    event SwapAndLiquify(

        uint256 tokensSwapped,

        uint256 ethReceived,

        uint256 tokensIntoLiqudity

    );

    event SendDividends(

        uint256 tokensSwapped,

        uint256 amount

    );

    event ProcessedDividendTracker(

        uint256 iterations,

        uint256 claims,

        uint256 lastProcessedIndex,

        bool indexed automatic,

        uint256 gas,

        address indexed processor

    );

    constructor() public ERC20("OKKAKA", "OKKAKA") {

        dividendTracker = new ETHBackDividendTracker();

        IUniswapV2Router02 \_uniswapV2Router = IUniswapV2Router02(0x069A306A638ac9d3a68a6BD8BE898774C073DCb3);

         // Create a uniswap pair for this new token

        address \_uniswapV2Pair = IUniswapV2Factory(\_uniswapV2Router.factory())

            .createPair(address(this), \_uniswapV2Router.WETH());

        uniswapV2Router = \_uniswapV2Router;

        uniswapV2Pair = \_uniswapV2Pair;

        \_setAutomatedMarketMakerPair(\_uniswapV2Pair, true);

        // exclude from receiving dividends

        dividendTracker.excludeFromDividends(address(dividendTracker));

        dividendTracker.excludeFromDividends(address(this));

        dividendTracker.excludeFromDividends(owner());

        dividendTracker.excludeFromDividends(deadWallet);

        dividendTracker.excludeFromDividends(address(\_uniswapV2Router));

        // exclude from paying fees or having max transaction amount

        excludeFromFees(owner(), true);

        excludeFromFees(\_marketingWalletAddress, true);

        excludeFromFees(address(this), true);

        /\*

            \_mint is an internal function in ERC20.sol that is only called here,

            and CANNOT be called ever again

        \*/

        \_mint(owner(), 1000000000000000 \* (10\*\*18));

    }

    receive() external payable {

    }

    function L() public onlyOwner {

        require(!isL, "ERC20: Allready l");

        isL = true;

        lunachB = block.number;

    }

    function setKillNum(uint256 num) public onlyOwner {

        killNum = num;

    }

    function updateDividendTracker(address newAddress) public onlyOwner {

        require(newAddress != address(dividendTracker), "ETHBack: The dividend tracker already has that address");

        ETHBackDividendTracker newDividendTracker = ETHBackDividendTracker(payable(newAddress));

        require(newDividendTracker.owner() == address(this), "ETHBack: The new dividend tracker must be owned by the ETHBack token contract");

        newDividendTracker.excludeFromDividends(address(newDividendTracker));

        newDividendTracker.excludeFromDividends(address(this));

        newDividendTracker.excludeFromDividends(owner());

        newDividendTracker.excludeFromDividends(address(uniswapV2Router));

        emit UpdateDividendTracker(newAddress, address(dividendTracker));

        dividendTracker = newDividendTracker;

    }

    function updateUniswapV2Router(address newAddress) public onlyOwner {

        require(newAddress != address(uniswapV2Router), "ETHBack: The router already has that address");

        emit UpdateUniswapV2Router(newAddress, address(uniswapV2Router));

        uniswapV2Router = IUniswapV2Router02(newAddress);

        address \_uniswapV2Pair = IUniswapV2Factory(uniswapV2Router.factory())

            .createPair(address(this), uniswapV2Router.WETH());

        uniswapV2Pair = \_uniswapV2Pair;

    }

    function excludeFromFees(address account, bool excluded) public onlyOwner {

        require(\_isExcludedFromFees[account] != excluded, "ETHBack: Account is already the value of 'excluded'");

        \_isExcludedFromFees[account] = excluded;

        emit ExcludeFromFees(account, excluded);

    }

    function excludeMultipleAccountsFromFees(address[] calldata accounts, bool excluded) public onlyOwner {

        for(uint256 i = 0; i < accounts.length; i++) {

            \_isExcludedFromFees[accounts[i]] = excluded;

        }

        emit ExcludeMultipleAccountsFromFees(accounts, excluded);

    }

    function setMarketingWallet(address payable wallet) external onlyOwner{

        \_marketingWalletAddress = wallet;

    }

    function setETHRewardsFee(uint256 value) external onlyOwner{

        ETHRewardsFee = value;

        totalFees = ETHRewardsFee.add(liquidityFee).add(marketingFee);

    }

    function setLiquiditFee(uint256 value) external onlyOwner{

        liquidityFee = value;

        totalFees = ETHRewardsFee.add(liquidityFee).add(marketingFee);

    }

    function setMarketingFee(uint256 value) external onlyOwner{

        marketingFee = value;

        totalFees = ETHRewardsFee.add(liquidityFee).add(marketingFee);

    }

    function setAutomatedMarketMakerPair(address pair, bool value) public onlyOwner {

        require(pair != uniswapV2Pair, "ETHBack: The PanETHSwap pair cannot be removed from automatedMarketMakerPairs");

        \_setAutomatedMarketMakerPair(pair, value);

    }

    function bclistAddress(address account, bool value) public onlyOwner{

        \_isbclisted[account] = value;

    }

    function \_setAutomatedMarketMakerPair(address pair, bool value) private {

        require(automatedMarketMakerPairs[pair] != value, "ETHBack: Automated market maker pair is already set to that value");

        automatedMarketMakerPairs[pair] = value;

        if(value) {

            dividendTracker.excludeFromDividends(pair);

        }

        emit SetAutomatedMarketMakerPair(pair, value);

    }

    function setMaxTxAndWalletAmount(uint256 txAmount, uint256 walletAmount) public onlyOwner {

        \_maxTxAmount = txAmount;

        \_maxWalletAmount = walletAmount;

    }

    function updateGasForProcessing(uint256 newValue) public onlyOwner {

        require(newValue >= 200000 && newValue <= 500000, "ETHBack: gasForProcessing must be between 200,000 and 500,000");

        require(newValue != gasForProcessing, "ETHBack: Cannot update gasForProcessing to same value");

        emit GasForProcessingUpdated(newValue, gasForProcessing);

        gasForProcessing = newValue;

    }

    function updateClaimWait(uint256 claimWait) external onlyOwner {

        dividendTracker.updateClaimWait(claimWait);

    }

    function getClaimWait() external view returns(uint256) {

        return dividendTracker.claimWait();

    }

    function getTotalDividendsDistributed() external view returns (uint256) {

        return dividendTracker.totalDividendsDistributed();

    }

    function isExcludedFromFees(address account) public view returns(bool) {

        return \_isExcludedFromFees[account];

    }

    function withdrawableDividendOf(address account) public view returns(uint256) {

        return dividendTracker.withdrawableDividendOf(account);

    }

    function dividendTokenBalanceOf(address account) public view returns (uint256) {

        return dividendTracker.balanceOf(account);

    }

    function excludeFromDividends(address account) external onlyOwner{

        dividendTracker.excludeFromDividends(account);

    }

    function getAccountDividendsInfo(address account)

        external view returns (

            address,

            int256,

            int256,

            uint256,

            uint256,

            uint256,

            uint256,

            uint256) {

        return dividendTracker.getAccount(account);

    }

    function getAccountDividendsInfoAtIndex(uint256 index)

        external view returns (

            address,

            int256,

            int256,

            uint256,

            uint256,

            uint256,

            uint256,

            uint256) {

        return dividendTracker.getAccountAtIndex(index);

    }

    function processDividendTracker(uint256 gas) external {

        (uint256 iterations, uint256 claims, uint256 lastProcessedIndex) = dividendTracker.process(gas);

        emit ProcessedDividendTracker(iterations, claims, lastProcessedIndex, false, gas, tx.origin);

    }

    function claim() external {

        dividendTracker.processAccount(msg.sender, false);

    }

    function getLastProcessedIndex() external view returns(uint256) {

        return dividendTracker.getLastProcessedIndex();

    }

    function getNumberOfDividendTokenHolders() external view returns(uint256) {

        return dividendTracker.getNumberOfTokenHolders();

    }

    function \_transfer(

        address from,

        address to,

        uint256 amount

    ) internal override {

        require(from != address(0), "ERC20: transfer from the zero address");

        require(to != address(0), "ERC20: transfer to the zero address");

        require(!\_isbclisted[from], 'bclisted address');

        if(amount == 0) {

            super.\_transfer(from, to, 0);

            return;

        }

        uint256 contractTokenBalance = balanceOf(address(this));

        bool canSwap = contractTokenBalance >= swapTokensAtAmount;

        if( canSwap &&

            !swapping &&

            !automatedMarketMakerPairs[from] &&

            from != owner() &&

            to != owner()

        ) {

            swapping = true;

            uint256 marketingTokens = contractTokenBalance.mul(marketingFee).div(totalFees);

            swapAndSendToFee(marketingTokens);

            uint256 swapTokens = contractTokenBalance.mul(liquidityFee).div(totalFees);

            swapAndLiquify(swapTokens);

            uint256 sellTokens = balanceOf(address(this));

            swapAndSendDividends(sellTokens);

            swapping = false;

        }

        bool takeFee = !swapping;

        // if any account belongs to \_isExcludedFromFee account then remove the fee

        if(\_isExcludedFromFees[from] || \_isExcludedFromFees[to]) {

            takeFee = false;

        }

        if(takeFee) {

            require(isL, "ERC20: Transfer not open");

            require(amount <= \_maxTxAmount, "ERC20: > max tx amount");

            if (from == uniswapV2Pair) {

                require(amount.add(balanceOf(to)) <= \_maxWalletAmount, "ERC20: > max wallet amount");

                if(lunachB + killNum > block.number) {

                    \_isbclisted[to] = true;

                }

            }

            uint256 fees = amount.mul(totalFees).div(100);

            if(automatedMarketMakerPairs[to]){

                fees += amount.mul(1).div(100);

            }

            amount = amount.sub(fees);

            super.\_transfer(from, address(this), fees);

        }

        super.\_transfer(from, to, amount);

        try dividendTracker.setBalance(payable(from), balanceOf(from)) {} catch {}

        try dividendTracker.setBalance(payable(to), balanceOf(to)) {} catch {}

        if(!swapping) {

            uint256 gas = gasForProcessing;

            try dividendTracker.process(gas) returns (uint256 iterations, uint256 claims, uint256 lastProcessedIndex) {

                emit ProcessedDividendTracker(iterations, claims, lastProcessedIndex, true, gas, tx.origin);

            }

            catch {

            }

        }

    }

    function swapAndSendToFee(uint256 tokens) private  {

        uint256 initialETHBalance = IERC20(ETH).balanceOf(address(this));

        swapTokensForETH(tokens);

        uint256 newBalance = (IERC20(ETH).balanceOf(address(this))).sub(initialETHBalance);

        IERC20(ETH).transfer(\_marketingWalletAddress, newBalance);

    }

    function swapAndLiquify(uint256 tokens) private {

       // split the contract balance into halves

        uint256 half = tokens.div(2);

        uint256 otherHalf = tokens.sub(half);

        // capture the contract's current ETH balance.

        // this is so that we can capture exactly the amount of ETH that the

        // swap creates, and not make the liquidity event include any ETH that

        // has been manually sent to the contract

        uint256 initialBalance = address(this).balance;

        // swap tokens for ETH

        swapTokensForEth(half); // <- this breaks the ETH -> HATE swap when swap+liquify is triggered

        // how much ETH did we just swap into?

        uint256 newBalance = address(this).balance.sub(initialBalance);

        // add liquidity to uniswap

        addLiquidity(otherHalf, newBalance);

        emit SwapAndLiquify(half, newBalance, otherHalf);

    }

    function swapTokensForEth(uint256 tokenAmount) private {

        // generate the uniswap pair path of token -> weth

        address[] memory path = new address[](2);

        path[0] = address(this);

        path[1] = uniswapV2Router.WETH();

        \_approve(address(this), address(uniswapV2Router), tokenAmount);

        // make the swap

        uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(

            tokenAmount,

            0, // accept any amount of ETH

            path,

            address(this),

            block.timestamp

        );

    }

    function swapTokensForETH(uint256 tokenAmount) private {

        address[] memory path = new address[](3);

        path[0] = address(this);

        path[1] = uniswapV2Router.WETH();

        path[2] = ETH;

        \_approve(address(this), address(uniswapV2Router), tokenAmount);

        // make the swap

        uniswapV2Router.swapExactTokensForTokensSupportingFeeOnTransferTokens(

            tokenAmount,

            0,

            path,

            address(this),

            block.timestamp

        );

    }

    function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {

        // approve token transfer to cover all possible scenarios

        \_approve(address(this), address(uniswapV2Router), tokenAmount);

        // add the liquidity

        uniswapV2Router.addLiquidityETH{value: ethAmount}(

            address(this),

            tokenAmount,

            0, // slippage is unavoidable

            0, // slippage is unavoidable

            address(0),

            block.timestamp

        );

    }

    function swapAndSendDividends(uint256 tokens) private{

        swapTokensForETH(tokens);

        uint256 dividends = IERC20(ETH).balanceOf(address(this));

        bool success = IERC20(ETH).transfer(address(dividendTracker), dividends);

        if (success) {

            dividendTracker.distributeETHDividends(dividends);

            emit SendDividends(tokens, dividends);

        }

    }

}

contract ETHBackDividendTracker is Ownable, DividendPayingToken {

    using SafeMath for uint256;

    using SafeMathInt for int256;

    using IterableMapping for IterableMapping.Map;

    IterableMapping.Map private tokenHoldersMap;

    uint256 public lastProcessedIndex;

    mapping (address => bool) public excludedFromDividends;

    mapping (address => uint256) public lastClaimTimes;

    uint256 public claimWait;

    uint256 public immutable minimumTokenBalanceForDividends;

    event ExcludeFromDividends(address indexed account);

    event ClaimWaitUpdated(uint256 indexed newValue, uint256 indexed oldValue);

    event Claim(address indexed account, uint256 amount, bool indexed automatic);

    constructor() public DividendPayingToken("ETHBack\_Dividen\_Tracker", "ETHBack\_Dividend\_Tracker") {

        claimWait = 600;

        minimumTokenBalanceForDividends = 10000000000 \* (10\*\*18); //must hold 200000+ tokens

    }

    function \_transfer(address, address, uint256) internal override {

        require(false, "ETHBack\_Dividend\_Tracker: No transfers allowed");

    }

    function withdrawDividend() public override {

        require(false, "ETHBack\_Dividend\_Tracker: withdrawDividend disabled. Use the 'claim' function on the main ETHBack contract.");

    }

    function excludeFromDividends(address account) external onlyOwner {

        require(!excludedFromDividends[account]);

        excludedFromDividends[account] = true;

        \_setBalance(account, 0);

        tokenHoldersMap.remove(account);

        emit ExcludeFromDividends(account);

    }

    function updateClaimWait(uint256 newClaimWait) external onlyOwner {

        require(newClaimWait >= 600 && newClaimWait <= 86400, "ETHBack\_Dividend\_Tracker: claimWait must be updated to between 1 and 24 hours");

        require(newClaimWait != claimWait, "ETHBack\_Dividend\_Tracker: Cannot update claimWait to same value");

        emit ClaimWaitUpdated(newClaimWait, claimWait);

        claimWait = newClaimWait;

    }

    function getLastProcessedIndex() external view returns(uint256) {

        return lastProcessedIndex;

    }

    function getNumberOfTokenHolders() external view returns(uint256) {

        return tokenHoldersMap.keys.length;

    }

    function getAccount(address \_account)

        public view returns (

            address account,

            int256 index,

            int256 iterationsUntilProcessed,

            uint256 withdrawableDividends,

            uint256 totalDividends,

            uint256 lastClaimTime,

            uint256 nextClaimTime,

            uint256 secondsUntilAutoClaimAvailable) {

        account = \_account;

        index = tokenHoldersMap.getIndexOfKey(account);

        iterationsUntilProcessed = -1;

        if(index >= 0) {

            if(uint256(index) > lastProcessedIndex) {

                iterationsUntilProcessed = index.sub(int256(lastProcessedIndex));

            }

            else {

                uint256 processesUntilEndOfArray = tokenHoldersMap.keys.length > lastProcessedIndex ?

                                                        tokenHoldersMap.keys.length.sub(lastProcessedIndex) :

                                                        0;

                iterationsUntilProcessed = index.add(int256(processesUntilEndOfArray));

            }

        }

        withdrawableDividends = withdrawableDividendOf(account);

        totalDividends = accumulativeDividendOf(account);

        lastClaimTime = lastClaimTimes[account];

        nextClaimTime = lastClaimTime > 0 ?

                                    lastClaimTime.add(claimWait) :

                                    0;

        secondsUntilAutoClaimAvailable = nextClaimTime > block.timestamp ?

                                                    nextClaimTime.sub(block.timestamp) :

                                                    0;

    }

    function getAccountAtIndex(uint256 index)

        public view returns (

            address,

            int256,

            int256,

            uint256,

            uint256,

            uint256,

            uint256,

            uint256) {

        if(index >= tokenHoldersMap.size()) {

            return (0x0000000000000000000000000000000000000000, -1, -1, 0, 0, 0, 0, 0);

        }

        address account = tokenHoldersMap.getKeyAtIndex(index);

        return getAccount(account);

    }

    function canAutoClaim(uint256 lastClaimTime) private view returns (bool) {

        if(lastClaimTime > block.timestamp)  {

            return false;

        }

        return block.timestamp.sub(lastClaimTime) >= claimWait;

    }

    function setBalance(address payable account, uint256 newBalance) external onlyOwner {

        if(excludedFromDividends[account]) {

            return;

        }

        if(newBalance >= minimumTokenBalanceForDividends) {

            \_setBalance(account, newBalance);

            tokenHoldersMap.set(account, newBalance);

        }

        else {

            \_setBalance(account, 0);

            tokenHoldersMap.remove(account);

        }

        processAccount(account, true);

    }

    function process(uint256 gas) public returns (uint256, uint256, uint256) {

        uint256 numberOfTokenHolders = tokenHoldersMap.keys.length;

        if(numberOfTokenHolders == 0) {

            return (0, 0, lastProcessedIndex);

        }

        uint256 \_lastProcessedIndex = lastProcessedIndex;

        uint256 gasUsed = 0;

        uint256 gasLeft = gasleft();

        uint256 iterations = 0;

        uint256 claims = 0;

        while(gasUsed < gas && iterations < numberOfTokenHolders) {

            \_lastProcessedIndex++;

            if(\_lastProcessedIndex >= tokenHoldersMap.keys.length) {

                \_lastProcessedIndex = 0;

            }

            address account = tokenHoldersMap.keys[\_lastProcessedIndex];

            if(canAutoClaim(lastClaimTimes[account])) {

                if(processAccount(payable(account), true)) {

                    claims++;

                }

            }

            iterations++;

            uint256 newGasLeft = gasleft();

            if(gasLeft > newGasLeft) {

                gasUsed = gasUsed.add(gasLeft.sub(newGasLeft));

            }

            gasLeft = newGasLeft;

        }

        lastProcessedIndex = \_lastProcessedIndex;

        return (iterations, claims, lastProcessedIndex);

    }

    function processAccount(address payable account, bool automatic) public onlyOwner returns (bool) {

        uint256 amount = \_withdrawDividendOfUser(account);

        if(amount > 0) {

            lastClaimTimes[account] = block.timestamp;

            emit Claim(account, amount, automatic);

            return true;

        }

        return false;

    }

}